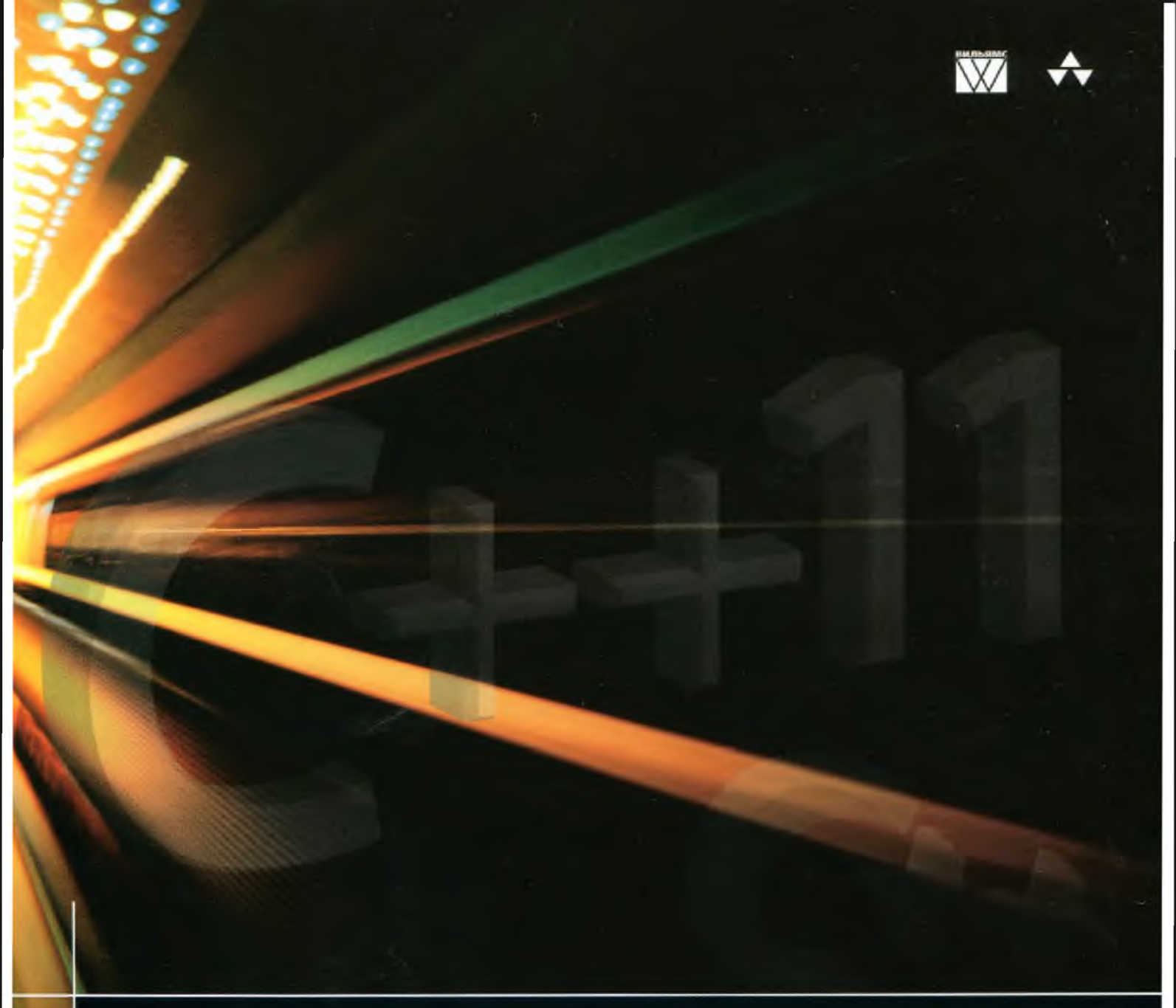


СТАНДАРТНАЯ БИБЛИОТЕКА C++

Справочное руководство

ВТОРОЕ ИЗДАНИЕ

НИКОЛАИ М. ДЖОСАТТИС

СТАНДАРТНАЯ БИБЛИОТЕКА C++

Справочное руководство

Второе издание

THE C++ STANDARD LIBRARY

A Tutorial and Reference

Second Edition

NICOLAI M. JOSUTTIS

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

СТАНДАРТНАЯ БИБЛИОТЕКА C++

Справочное руководство

Второе издание

НИКОЛАИ М. ДЖОСАТТИС

Москва · Санкт-Петербург · Киев
2014

ББК 32.973.26-018.2.75
Д42
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция докт. физ.-мат. наук *Д.А. Клюшина*
Консультант канд. техн. наук *И.В. Красиков*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Джосаттис, Николаи М.

Д42 Стандартная библиотека C++: справочное руководство, 2-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2014. — 1136 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1837-6 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Николаи М. Джосаттис

Стандартная библиотека C++: справочное руководство

2-е издание

Литературный редактор *И.А. Попова*
Верстка *О.В. Мишутина*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 23.06.2014. Формат 70x100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 91,59. Уч.-изд. л. 58,7
Тираж 1500 экз. Заказ № 3214.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9-я линия, 12/28

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1837-6 (рус.)
ISBN 978-0-321-62321-8 (англ.)

Оглавление

Содержание

Предисловие ко второму изданию

Я никогда не предполагал, что первое издание книги будет продаваться так долго. Однако сейчас, по прошествии двенадцати лет, настало время для нового издания, которое отражало бы C++11 — новый стандарт языка C++.

Эта задача не сводится к простому добавлению новых библиотек. Язык C++ изменился. Почти все типичные приложения стандартной библиотеки теперь выглядят немного по-другому. Это следствие не одного крупного изменения языка, а множества мелких модификаций, таких как использование rvalue-ссылок и семантика перемещения, циклов `for` по коллекциям, ключевое слово `auto` и новые свойства шаблонов. Таким образом, помимо описания новых библиотек и дополнительных функциональных возможностей существующих библиотек, в книге пришлось переписать (хотя бы частично) практически все примеры. Тем не менее для поддержки программистов, использующих "старые" инструментальные среды для языка C++, в книге описываются различия между версиями языка C++.

Я тщательно изучил стандарт C++11. Поскольку я не следовал стандарту, то стал изучать документ C++11 около двух лет назад. Я действительно понимал его с трудом. Но члены комитета по стандартизации помогли мне описать и представить новые функциональные возможности.

В результате возникла проблема: несмотря на то, что размер книги вырос с 800 до более чем 1110 страниц (в англоязычном издании. — *Примеч. ред.*), представить стандартную библиотеку C++ полностью оказалось невозможным. Часть стандарта C++11, описывающая библиотеку, состоит из примерно 750 страниц, написанных очень лаконично, без дополнительных разъяснений. По этой причине я решил описать эти функциональные возможности намного подробнее. Принять это решение мне снова помогли многие члены комитета по стандартизации. Я хотел сосредоточиться на том, что необходимо среднестатистическим прикладным программистам. Пропущенные функциональные возможности я описал в дополнительной главе, размещенной в веб по адресу `http://www.cppstdlib.com`, но и после этого вы можете встретить в стандарте понятия, которые не описаны в книге.

Искусство преподавания не сводится к изложению всего без исключения. Это искусство отделения зерен от плевел. Надеюсь, что у меня это получилось.

Благодарности ко второму изданию

В книге представлены идеи, концепции, решения и примеры из многих источников. За последние несколько лет сообщество C++ выработало много идей, концепций, предложений и усовершенствований языка C++, ставших частью стандарта C++11. Таким образом, я хотел бы снова выразить свою благодарность всем людям, поддерживавшим меня в работе над новым изданием.

Во-первых, благодарю всех членов сообщества C++ и членов комитета по стандартизации. Помимо работы над новыми функциональными возможностями языка и библиотеки, они потратили много времени на объяснение мне многочисленных новшеств, причем делали они это терпеливо и увлеченно.

Скотт Мейерс (Scott Meyers) и Энтони Уильямс (Anthony Williams) разрешили мне использовать свои учебные материалы и рукописи книг, в которых я смог найти множество полезных и еще не опубликованных примеров.

Я бы хотел также выразить свою признательность всем, кто прислал отзывы на книгу и высказал свои пожелания и ценные советы: Дэйву Абрамсу (Dave Abrahams), Альберто Ганешу Барбати (Alberto Ganesh Barbati), Пит Беккеру (Pete Becker), Томасу Беккеру (Thomas Becker), Хансу Бёму (Hans Boehm), Вальтеру Е. Брауну (Walter E. Brown), Паоло Карлини (Paolo Carlini), Лоуренсу Кроулу (Lawrence Crowl), Беману Дэйвзу (Beman Dawes), Дугу Грэгору (Doug Gregor), Дэвиду Гризби (David Grigsby), Пабло Гальперну (Pablo Halpern), Ховарду Ханнанту (Howard Hinnant), Джону Лакосу (John Lakos), Броне ку Козицки (Bronek Kozicki), Дитмару Кюлю (Dietmar Kühl), Даниэлю Крюглеру (Daniel Krügler), Мэту Маркусу (Mat Marcus), Йенсу Мауреру (Jens Maurer), Алисдайру Мередиту (Alisdair Meredith), Бартошу Милевски (Bartosz Milewski), П.Дж. Плагеру (P. J. Plauger), Тобиасу Шюле (Tobias Schüle), Петеру Соммерланду (Peter Sommerlad), Джонатану Уэйкли (Jonathan Wakely) и Энтони Уильямсу (Anthony Williams).

Один человек сделал особенно выдающуюся работу. Если у меня возникал вопрос, Даниэль Крюглер отвечал практически немедленно с невероятной точностью и компетентностью. Каждый, кто занимался процессом стандартизации языка C++, знает, Даниэль общается так со всеми. Без него и стандарт языка C++ и эта книга были ба не такими качественными.

Я очень благодарен редакторам Петеру Гордону (Peter Gordon), Ким Бёдигхаймер (Kim Boedigheimer), Джону Фаллеру (John Fuller) и Анне Попик (Anna Popisck) из издательства Addison-Wesley. Кроме оказываемой ими поддержки, они нашли правильный баланс между терпением и давлением. технический редактор Эвелин Пайл (Evelyn Pyle) и корректор Дайан Фрид (Diane Freed) проделали невероятную работу по переводу моего немецкого английского на американский английский. Кроме того, я благодарю Фрэнка Миттельбаха (Frank Mittelbach) за решение проблем, связанных с пакетом LATEX.

И наконец, я безмерно признателен Ютте Экштейн (Jutta Eckstein). Ютта обладает чудесной способностью мотивировать и поддерживать людей в развитии их идей и достижении идеалов и целей. Многие люди испытывают ее влияние только в совместной работе, а мне посчастливилось изведать это в повседневной жизни.

Предисловие к первому изданию

Вначале я планировал написать лишь небольшую книгу на немецком языке (около 400 страниц) о стандартной библиотеке C++. Это было в 1993 году. Теперь, десять лет спустя, книга на английском языке содержит более 800 страниц текста, рисунков и примеров. Я намеревался описать стандартную библиотеку C++ так, чтобы читатель мог найти ответы на все (или почти все) вопросы еще до того, как они у него возникнут. Впрочем, книга не содержит полного описания всех аспектов стандартной библиотеки C++. Вместо этого я выбрал темы, которые кажутся мне самыми важными для изучения и программирования на языке C++ с использованием стандартной библиотеки.

Изложение всех тем следует одному общему принципу: общее описание завершается изложением конкретных деталей, необходимых для решения повседневных задач программирования. Примеры программ помогут читателю глубже понять принципы и освоить детали.

Вот, в принципе, и все. Надеюсь, что чтение этой книги принесет вам столько же удовольствия, сколько я получил, когда писал ее. Приятного чтения!

Благодарности к первому изданию

В книге изложены идеи, концепции, решения и примеры из многих источников. В некотором смысле даже несправедливо, что на обложке стоит только моя фамилия, поэтому я хотел бы поблагодарить всех людей и компании, помогавшие мне на протяжении последних нескольких лет.

Прежде всего я благодарен Дитмару Кюлю (Dietmar Kül), эксперту по языку C++ (особенно в области потоков ввода-вывода и интернационализации, — он реализовал библиотеку потоков ввода-вывода просто для развлечения). Дитмар не только перевел большие фрагменты книги с немецкого на английский, но и написал некоторые разделы на основе собственного опыта. Кроме того, все эти годы он давал мне неоценимые советы.

Далее я хочу поблагодарить всех рецензентов и других людей, высказавших свое мнение. Благодаря им книга имеет такой уровень качества, которого не могло бы быть без их поддержки. (Список получился довольно длинным, поэтому прошу меня простить, если я кого-то забыл.) Рецензентами английской версии книги были Чак Эллисон (Chuck Allison), Грег Комо (Greg Comeau), Джеймс А. Кротингер (James A. Crotinger), Гэбриел Дос Рейс (Gabriel Dos Reis), Алан Эзаст (Alan Ezust), Натан Майерс (Nathan Myers), Вернер Мосснер (Werner Mossner), Тодд Велдхузен (Todd Veldhuizen), Чичан Ван (Chichiang Wan), Джуди Вард (Judy Ward) и Томас Викхульт (Thomas Wikehult). Немецкую версию рецензировали Ральф Бёккер (Ralph Boecker), Дирк Херрманн (Dirk Herrmann), Дитмар Кюль (Dietmar Kühl), Эдда Лёрке (Edda Löke), Герберт Шойбнер (Herbert Scheubner), Доминик Штрассер (Dominik Strasser) и Мартин Вейцель (Martin Weitzel). Дополнительный вклад внесли Мэтт Остерн (Matt Austern), Валентин Боннард (Valentin Bonnard), Грег Колвин (Greg Kolvin), Беман Доуз (Beman Dawes), Билл Гиббонз (Bill Gibbons), Лоис Голдтвейт (Lois Goldthwaite), Эндрю Кёниг (Andrew Koenig), Стив Рамсби (Steve Rumsby), Бьярне Страуструп (Bjarne Stroustrup) и Дэвид Вандевурд (David Vandevoorde).

Я особенно благодарен Дэйву Абрамсу (Dave Abrahams), Джанет Кокер (Janet Cocker), Кэтрин Охала (Catherine Ohala) и Морин Уиллард (Maureen Willard) за внимательное рецензирование и редактирование всей книги. Их советы внесли неоценимый вклад в повышение ее качества.

Особую благодарность хочу выразить Гербу Саттеру (Herb Sutter), автору знаменитой серии задач по программированию на языке C++ — *Guru of the Week*, — публикуемой в группе новостей `comp.lang.c++.moderated`.

Я также благодарен всем людям и компаниям, помогавшим мне в тестировании примеров на разных платформах и с разными компиляторами. Большое спасибо Стиву Адамчику (Steve Adamczyk), Майку Андерсону (Mike Anderson) и Джону Спайсеру (John Spicer) из компании EDG за отличный компилятор и техническую поддержку. Они очень помогли при стандартизации и написании книги. Я благодарен П. Дж. Плаугеру (P. J. Plauger) и компании Dinkumware, Ltd. за их раннюю реализацию стандартной библиотеки C++. Спасибо Андреасу Хоммелю (Andreas Hommel) и компании Metrowerks за пробную версию среды CodeWarrior Programming Environment. Большое спасибо разработчикам бесплатных компиляторов GNU и egcs. Огромное спасибо компании Microsoft за пробную версию интегрированной среды Visual C++. Я благодарен Роланду Хартингеру (Roland Hartinger) из компании Siemens Nixdorf Information Systems AG за их пробную версию компилятора C++. Спасибо компании Topjects GmbH за пробную версию библиотеки ObjectSpace.

Большое спасибо сотрудникам издательства Addison Wesley Longman, работавшим со мной. В числе прочих хочу особо поблагодарить Дженет Кокер (Janet Cocker), Майка Хендриксона (Mike Hendrickson), Дебби Лафферти (Debbie Lafferty), Марину Ланг (Marina

Lang), Чанду Лири (Chanda Leary), Кэтрин Охала (Catherine Ohala), Марти Рабинович (Marty Rabinowitz), Сьюзен Спицер (Susan Spitzer) и Морин Уиллард (Maureen Willard). С вами было приятно работать.

Кроме того, я благодарен сотрудникам компании BREDEX GmbH и всем членам сообщества C++, особенно участникам процесса стандартизации, за поддержку и терпение (иногда я задавал действительно глупые вопросы).

И наконец, я сердечно благодарю мою семью: Улли, Лукаса, Анику и Фредерика. Я определенно не уделял им должного внимания во время работы над книгой.

Спасибо всем!

Ждем ваших отзывов!

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 интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сервер и оставить свои замечания там. 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 интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@dialektika.com
WWW: http://www.dialektika.com

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1
в Украине: 03150, Киев, а/я 152

Глава 1

О книге

1.1. Для чего предназначена эта книга

Вскоре после своего появления язык C++ стал фактическим стандартом объектно-ориентированного программирования. Вследствие этого появилась необходимость в стандартизации. Только имея общепринятый стандарт, можно писать программы, работающие на разных платформах — от персональных компьютеров до мейнфреймов. Более того, стандартная *библиотека* позволила бы программистам использовать универсальные компоненты и обеспечила бы более высокий уровень абстрактности без потери переносимости программ, избавив программистов от необходимости разрабатывать все программы с самого начала.

Теперь, с появлением второго стандарта, получившего название C++11 (см. раздел 2.1, в котором описана подробная история стандартов языка C++), мы получили в свое распоряжение огромную библиотеку языка C++, спецификация которой вдвое больше, чем описания основных средств языка. Стандартная библиотека состоит из следующих компонентов.

- Классы ввода-вывода (I/O).
- Типы строк и регулярные выражения.
- Разные структуры данных, такие как динамические массивы, связанные списки, бинарные деревья и хеш–таблицы.
- Многочисленные алгоритмы, например, множество алгоритмов сортировки.
- Классы для многопоточной и параллельной работы.
- Классы для поддержки интернационализации.
- Числовые классы.
- Огромное количество утилит.

Однако стандартная библиотека не является самоочевидной. Для того чтобы использовать ее компоненты и пользоваться их преимуществами, сначала необходимо разобраться с основными концепциями и важными подробностями, — простое перечисление классов и их функций здесь не поможет. Именно для этого и была написана эта книга. В первой части книги библиотека и все ее компоненты описываются на концептуальном уровне, а затем излагаются подробности, необходимые для практического программирования. Для демонстрации использования всех компонентов в книгу включены примеры. Таким образом, книга представляет собой подробное введение в библиотеку C++ как для начинающих, так

и для опытных программистов. Обладая изложенной в книге информацией, вы сможете в полной мере использовать преимущества стандартной библиотеки C++.

Тем не менее следует заметить, что далеко не все содержание книги является простым и самоочевидным. Стандартная библиотека обеспечивает высокую гибкость, однако в нетривиальных ситуациях гибкость создает побочные эффекты. В библиотеке есть ловушки и просчеты. Я их укажу и предложу способ обойти их.

1.2. Что необходимо знать читателю

Для того чтобы понять бо́льшую часть книги, читатель должен знать язык C++. (В книге описываются стандартные компоненты C++, но не сам язык.) Читатель должен быть знаком с концепциями классов, наследования, шаблонов, обработки исключений и пространства имен. Однако от читателя не требуется досконального знания всех тонкостей языка. Самые важные детали описаны в книге, а тонкости имеют значение для программистов, занимающихся реализацией библиотеки, а не ее использованием. Следует помнить, что язык в процессе выработки стандарта C++11 подвергся изменениям, как это происходило при создании стандарта C++98, поэтому некоторые ваши знания могут оказаться устаревшими. В главе 3 приведен краткий обзор и описание новейших средств языка, необходимых для использования библиотеки C++11. Многие функциональные возможности новой библиотеки опираются на новые средства языка C++, поэтому главу 3 следует изучить внимательно. При описании компонентов библиотеки, использующих новые средства языка, я буду постоянно ссылаться на эту главу.

1.3. Стиль и структура книги

Компоненты стандартной библиотеки C++ так или иначе зависят друг от друга, поэтому их трудно описывать по отдельности. Существует несколько возможных вариантов изложения материала. Например, можно было бы придерживаться порядка изложения, принятого в стандарте C++. Однако описывать компоненты библиотеки C++ каждый раз с нуля — не лучшее решение. Можно было бы начать книгу обзором всех компонентов, а в последующих главах привести их подробное описание. В качестве альтернативы можно было бы попытаться определить порядок изложения с минимальным количеством перекрестных ссылок. В итоге мы решили использовать сочетание всех трех способов. Книга начинается с краткого изложения основных концепций и вспомогательных средств библиотеки. Далее следуют описания всех компонентов, каждое из которых занимает одну или несколько глав. Первым компонентом является стандартная библиотека шаблонов (Standard Template Library — STL). Несомненно, STL — самая мощная, самая сложная и самая интересная часть стандартной библиотеки C++. Ее структура сильно влияет на другие компоненты. Затем описываются более самоочевидные компоненты, например специализированные контейнеры, строки и регулярные выражения. Следующий компонент — библиотека IOStream — вероятно. хорошо знаком читателям. Затем обсуждаются вопросы интернационализации, оказывающие определенное влияние на библиотеку

IOStream. В заключение описываются части библиотеки, относящиеся к числам, параллельной работе и распределителям памяти.

Описание каждого компонента начинается с указания его предназначения, представления его структуры и примеров. Далее подробно рассматриваются разные способы использования компонента, а также ловушки и проблемы, которые при этом возникают. Описание обычно заканчивается справочным разделом, в котором приводятся точные сигнатуры и определения классов компонента и их функций.

Содержание книги

Первые пять глав представляют собой введение в стандартную библиотеку C++ в целом.

- **Глава 1, "О книге"**, описывает тему книги и ее содержание.
- **Глава 2, "Введение в язык C++ и стандартную библиотеку"**, содержит краткое изложение истории стандартной библиотеки C++ в контексте ее стандартизации. Кроме того, в ней вводится концепция сложности.
- **Глава 3, "Новые средства языка"**, содержит описание новых средств языка, необходимых для чтения книги и использования стандартной библиотеки C++.
- **Глава 4, "Общие принципы"**, описывает основные принципы библиотеки, которые необходимо понимать, чтобы использовать все ее компоненты. В частности, в этой главе представлено пространство имен `std`, формат заголовочных файлов и общие средства обработки ошибок и исключений.
- **Глава 5, "Вспомогательные средства"**, посвящена небольшим утилитам, предназначенным как для функционирования библиотеки, так и для ее пользователей. В частности, в этой главе описаны классы `pair<>` и `tuple<>`, интеллектуальные указатели, числовые диапазоны, свойства типов и утилиты для работы с типами, вспомогательные функции, класс `ratio<>`, часы и таймеры, а также доступные функции из языка C.

В главах 6–11 описаны все аспекты библиотеки STL.

- **Глава 6, "Стандартная библиотека шаблонов"**, содержит описание общих концепций библиотеки STL, содержащей контейнеры и алгоритмы, используемые для обработки коллекций данных. В главе подробно изложены концепция, проблемы и специальные приемы программирования с использованием STL, а также роли их компонентов.
- **Глава 7, "Контейнеры STL"**, посвящена контейнерным классам STL. В главе описываются массивы, векторы, деки, списки, форвардные списки, множества, ассоциативные массивы и неупорядоченные контейнеры, а также их общие возможности, различия между ними, конкретные преимущества и недостатки с иллюстративными примерами.
- **Глава 8, "Детальное описание контейнеров STL"**, содержит перечисление и описание всех членов контейнеров (т.е. типы и операции), представляя собой удобный справочник.
- **Глава 9, "Итераторы STL"**, подробное описывает категории итераторов STL, вспомогательные функции и адаптеры, такие как потоковые итераторы, обратные итераторы, итераторы вставки и итераторы перемещения.

- **Глава 10, "Функциональные объекты STL и лямбда-функции"**, посвящена классам функциональных объектов, STL, включая лямбда-функции и выражения. В ней также показано, как с их помощью определить поведение контейнеров и алгоритмов.
- **Глава 11, "Алгоритмы STL"**, содержит перечисление и описание алгоритмов STL. После краткого вступления и сравнения алгоритмов приводятся их подробные описания и примеры.

Главы 12–14 посвящены "простым" стандартным классам библиотеки C++.

- **Глава 12, "Специальные контейнеры"**, описывает адаптеры контейнеров для очередей и стеков, а также класс `bitset`, предназначенный для управления битовыми полями произвольной разрядности, или флагами.
- **Глава 13, "Строки"**, описывает типы строк в стандартной библиотеке C++ (да, их несколько). В стандарте строки называются "очевидными" фундаментальными типами данных, предоставляющими возможность использования разнообразных типов символов.
- **Глава 14, "Регулярные выражения"**, посвящена интерфейсам, предназначенным для работы с регулярными выражениями, которые используются для поиска и замены символов и подстрок.

В главах 15 и 16 рассматриваются тесно связанные друг с другом темы: ввод-вывод и интернационализация.

- **Глава 15, "Классы потоков ввода-вывода"**, содержит описание стандартизированной формы широко известной библиотеки IOStream. В главе также приводятся подробности, которые мало известны, но играют важную роль в программировании, например, правильный способ определения и интеграции специальных каналов ввода-вывода.
- **Глава 16, "Интернационализация"**, посвящена основным концепциям и классам, предназначенным для интернационализации программ, в частности, разным системам кодирования, а также форматам дат чисел с плавающей точкой.

В последней части описаны числовые классы, параллельная работа и распределители памяти.

- **Глава 17, "Работа с числами"**, посвящена числовым компонентам стандартной библиотеки C++: классам случайных чисел и распределений, типам комплексных чисел и некоторым числовым функциям из языка C.
- **Глава 18, "Параллельное программирование"**, описывает средства стандартной библиотеки C++ для обеспечения параллельной работы и многопоточного режима.
- **Глава 19, "Распределители памяти"**, посвящена моделям памяти в стандартной библиотеке C++.

Книга завершается **библиографией** и **предметным указателем**.

Из-за ограниченного объема книги я разместил материал, не предназначенный для среднестатистического программиста, в дополнительной главе, которая находится на веб-сайте `http://www.cppstdlib.com`. Эта глава посвящена следующим темам.

- Подробное описание битовых множеств (упомянуты в разделе 12.5).
- Класс `vallaray<>` (очень кратко описаны в разделе 17.4).
- Подробное описание распределителей памяти (упомянуты в главе 19).

1.4. Как читать книгу

Книга представляет собой сочетание руководства пользователя и справочника по стандартной библиотеке C++. Отдельные компоненты стандартной библиотеки C++ в некотором смысле не зависят друг от друга, поэтому после глав 2–5 главы, посвященные отдельным компонентам, можно читать в любом порядке. Следует помнить, что главы 6–11 описывают один и тот же компонент. Для того чтобы понять содержание остальных глав, посвященных библиотеке STL, начните с введения в STL (глава 6).

Если вы программируете на языке C++ и хотите освоить основные принципы и все компоненты стандартной библиотеки, то можете просто читать эту книгу от начала до конца. В этом случае справочные разделы можно пропустить. При использовании некоторых компонентов стандартной библиотеки C++ нужную информацию лучше всего искать по предметному указателю, который сделан достаточно подробным, чтобы минимизировать время поиска.

Мой опыт подсказывает, что новый материал лучше изучать на конкретных примерах. По этой причине в книгу включены многочисленные примеры разной величины: от нескольких строк до законченных программ. В последнем случае имя файла с программой указано в первой строке комментария. Файлы примеров можно загрузить с веб-сайта `http://www.cppstdlib.com`.

1.5. Последние достижения

Стандарт C++11 был завершен, когда я еще писал книгу. Пожалуйста, не забывайте, что некоторые компиляторы могут пока не поддерживать новый стандарт. В ближайшем будущем эта ситуация изменится. В результате вы можете обнаружить, что не вся информация, изложенная в книге, точно соответствует вашему компилятору, и некоторые примеры, возможно, придется уточнить с учетом конкретного программного окружения.

1.6. Примеры и дополнительная информация

Все примеры и дополнительную информацию о книге и стандартной библиотеке C++ можно найти на веб-сайте: `http://www.cppstdlib.com`. Кроме того, много дополнительной информации можно найти в Интернете. В библиографии указаны некоторые веб-сайты, которые могут оказаться полезными.

1.7. Обратная связь

Я предлагаю читателям высказывать свое мнение о книге (как положительное, так и отрицательное). Я старался написать ее как можно более тщательно; однако я — всего лишь человек и в какой-то момент должен был остановить работу над книгой. По этой причине в ней могут быть ошибки, неточности или неудачные выражения. Ваши замечания позволят мне устранить их в будущих изданиях.

Проще всего связаться со мной по электронной почте. Однако, для того чтобы не столкнуться с рассылкой спама, я не стал включать адрес своей электронной почты в эту книгу. (Я перестал пользоваться электронным адресом, указанным в первом издании книги, после того, как стал получать на него тысячи спам-сообщений каждый день.) Для того чтобы узнать адрес моей электронной почты, пожалуйста, зайдите на веб-сайт `http://www.cppstdlib.com`.

Большое спасибо.

Глава 2

Введение в язык C++ и стандартную библиотеку

В главе обсуждаются история и разные версии языка C++, а также вводится система *О-обозначения*, которая используется для оценки быстродействия и масштабируемости библиотечных операций.

2.1. История стандартов языка C++

Стандартизация языка C++ была начата в 1989 году Международной организацией по стандартизации (International Organization for Standardization — ISO), представляющей собой группу национальных организаций по стандартам, таких как ANSI в США. К настоящему моменту эта работа привела к выработке четырех вариантов стандарта C++, в той или иной степени доступных на разных платформах в разных странах.

1. **Документ C++98**, принятый в 1998 г., был первым стандартом языка C++. Его официальное название: Information Technology — Programming Languages — C++. Этот документ имеет номер ISO/IEC 14882:1998.
2. **Документ C++03** содержит "технические поправки" ("technical corrigendum" — "TC") ошибок, обнаруженных в стандарте C++98. Он имеет номер ISO/IEC 14882:2003. Таким образом, документы C++98 и C++03 образуют "первый стандарт языка C++".
3. **Документ TR1** содержит расширения библиотеки из первого стандарта. Его официальное название: *Information Technology — Programming Languages — Technical Report on C++ Library Extensions*. Он имеет номер ISO/IEC TR 19768:2007. Все внесенные изменения содержатся в пространстве имен `std::tr1`.
4. **Документ C++11**, принятый в 2011 г., является вторым стандартом языка C++. Он предусматривает значительные улучшения как языка, так и библиотеки, при этом расширения из документа TR1 стали частью пространства имен `std`). Официальное название документа совпадает с названием первого стандарта: *Information Technology — Programming Languages — C++*, но он имеет другой номер: ISO/IEC 14882:2011.

В книге описывается стандарт C++11, который в процессе работы над ним назывался "C++0x," поскольку ожидалось, что он будет принят не позднее 2009 года[1]. Таким образом, названия C++11 и C++0x относятся к одному и тому же документу. Во всей книге используется термин C++11.

Поскольку некоторые платформы и интегрированные среды еще не поддерживают полностью стандарт C++11 (как средства языка, так и функциональные возможности библиотек), новшества, появившиеся в новом стандарте, будут указаны отдельно.

[1] Программисты шутят, что символ x в конце концов стал шестнадцатеричным b.

2.1.1. Обычные вопросы о стандарте C++11

Где взять стандарт?

Самая свежая версия черновика стандарта C++11 доступна в виде документа N3242 (см. [C++Std2011Draft]). Этого черновика вполне достаточно для большинства пользователей и программистов, а те, кому нужен действительный стандарт, должны купить его у организации ISO или у национального агентства.

Почему стандартизация шла так долго?

У читателей может возникнуть вопрос: почему выработка обоих стандартов заняла больше десяти лет и почему в итоге процесс стандартизации не считается завершенным? Следует подчеркнуть, что стандарт является результатом работы многих людей и компаний, предложивших свои улучшения и расширения. В процессе стандартизации эти люди общались друг с другом, тестировали предложения и решали проблемы, возникающие при совмещении всех новшеств. Никто из них не был занят разработкой нового стандарта языка C++ полный рабочий день. Этот стандарт не является результатом работы компании с большим бюджетом и массой доступного времени. Организации по стандартизации ничего или почти ничего не платят людям, принимающим участие в разработке стандартов. Таким образом, если участник стандартизации не работал в компании, имевшей особый интерес в разработке стандарта, то он работал просто для удовольствия. К счастью, многие люди, разрабатывавшие стандарт языка C++, имели время и деньги, чтобы сделать это. Примерно пятьдесят–сто человек регулярно встречались примерно три раза в год, чтобы обсудить все вопросы и завершить работу, а в течение остального времени использовали электронную почту. Таким образом, не следовало ожидать совершенного и тщательно продуманного документа. Результат вполне применим на практике, но не идеален (впрочем, как и все на свете).

Описание стандартной библиотеки занимало примерно половину первого стандарта, а во втором стандарте ее доля возросла до 65%. (В документе C++11 количество страниц, посвященных библиотеке, возросло с 350 до 750.)

Отметим, что стандарт имеет разные источники. Фактически любая компания или страна и даже отдельные люди могли предложить свои новшества и функциональные возможности, которые потом должны были быть одобрены организацией по стандартизации. В принципе, ни один компонент не разрабатывался с нуля[2]. Таким образом, результат получился не очень однородным. Разные компоненты основаны на разных принципах проектирования. Ярким примером таких различий являются классы строк и библиотека STL, ставшая каркасом для структур данных и алгоритмов.

- Классы строк задумывались как безопасный и удобный компонент. По этой причине они обеспечивают практически самоочевидный интерфейс и проверяют многие ошибки в интерфейсе.
- Библиотека STL разрабатывалась для того, чтобы объединить разные структуры данных с разными алгоритмами и достичь повышенной производительности. По этой причине библиотека STL не очень удобна для использования и не предусматривает

[2] Читатели могут поинтересоваться, почему процесс стандартизации не предусматривал разработку библиотеки заново. Дело в том, что основная цель стандартизации — не изобретение или разработка чего-то нового, а гармонизация существующей практики.

проверки многих логических ошибок. Для того чтобы воспользоваться преимуществами библиотеки STL, программист должен знать ее концепции и правильно их применять.

Оба этих компонента являются частями одной и той же библиотеки. Они в определенной степени согласованы, но подчиняются своей собственной логике.

Тем не менее одной из целей стандарта C++11 было упрощение. По этой причине в стандарт C++11 было включено много предложений, направленных на решение проблем, устранение несогласованности и облегчение практической работы. Например, способ инициализации значений и объектов в стандарте C++11 гармонизирован. Кроме того, класс интеллектуальных указателей `auto_ptr` был заменен многочисленными и усовершенствованными классами интеллектуальных указателей, ранее опубликованными на вебсайте Boost, посвященном свободной, рецензируемой и машинонезависимой библиотеке исходных кодов на языке C++ (см. [Boost]). Это было сделано для того, чтобы получить практический опыт до включения этих классов в новый стандарт или технические поправки.

Является ли новый стандарт последним?

Стандарт C++11 не является окончательным. Люди постоянно исправляют ошибки, выдвигают дополнительные требования и предлагают новые средства. Таким образом, вполне вероятно, что рано или поздно появятся "технические поправки", в которых будут исправлены ошибки и устранены неточности. Это может быть документ "TR2" и/или третий стандарт.

2.1.2. Совместимость стандартов C++98 и C++11

Стандарт C++11 должен был сохранить обратную совместимость со стандартом C++98. В принципе, все программы, которые компилируются в соответствии со стандартами C++98 или C++03, должны компилироваться по стандарту C++11. Однако существуют исключения. Например, имена переменных не могут совпадать с новыми ключевыми словами.

Если код должен работать в разных версиях языка C++, используя при этом преимущества стандарта C++11, то можно использовать заранее определенный макрос `__cplusplus`. Следующая директива определяет использование стандарта C++11 при компиляции единицы трансляции на языке C++:

```
#define __cplusplus 201103L
```

В противоположность этому для использования стандартов C++98 и C++03 необходимо выполнить директиву

```
#define __cplusplus 199711L
```

Однако иногда поставщики компиляторов предусматривают в этих директивах другие значения.

Следует отметить, что обратная совместимость сохраняется только на уровне исходного кода. Бинарная совместимость не гарантируется, что порождает проблемы, особенно если существующая операция получает новый тип возвращаемого значения, поскольку перегрузка типа возвращаемого значения не допускается (например, это относится

к алгоритмам и некоторым функциям-членам контейнеров из библиотеки STL). Таким образом, компиляция всех частей программы, написанной по стандарту C++98, включая библиотеки, с помощью компилятора, поддерживающего стандарт C++11, не должна вызывать трудностей. Однако редактирование связей между кодом, скомпилированным по стандарту C++11, и кодом, созданным с помощью компилятора, поддерживающего стандарт C++98, может завершиться неудачно.

2.2. Сложность и O-обозначения

Для некоторых частей стандартной библиотеки языка C++ — особенно библиотеки STL — быстродействие алгоритма и функций-членов имеет большое значение. По этой причине стандарт предусматривает оценку их *сложности*. Для описания относительной сложности алгоритмов специалисты по компьютерным наукам используют специальное *O*-обозначение. С помощью этого обозначения они быстро определяют относительную продолжительность выполнения алгоритма, а также проводят качественное сравнение алгоритмов.

O-обозначение выражает продолжительность выполнения алгоритма в виде функции, зависящей от размера входного аргумента n. Например, если продолжительность выполнения растет линейно при росте количества элементов, т.е. удваивается при удвоении размера входного аргумента, сложность составляет O(n). Если продолжительность выполнения не зависит от размера входного аргумента, то сложность составляет O(1). Типичные значения сложности и их выражения с помощью символа O представлены в табл. 2.1.

Таблица 2.1. Типичные значения сложности

Тип сложности	Обозначение	Смысл
Константная	O(1)	Продолжительность выполнения не зависит от количества элементов.
Логарифмическая	O($log(n)$)	Продолжительность выполнения растет как логарифмическая функция в зависимости от количества элементов
Линейная	O(n)	Продолжительность выполнения линейно зависит (с постоянным коэффициентом) от количества элементов
n-log-n	O($n\ log(n)$)	Продолжительность выполнения зависит от количества элементов как функция, представляющая собой произведение линейной и логарифмической сложности
Квадратичная	O(n^2)	Продолжительность выполнения квадратично зависит от количества входных элементов

Следует подчеркнуть, что O-обозначение не учитывает множители с меньшей степенью, например константы. В частности, не имеет значения, как долго будет выполняться алгоритм. С этой точки зрения два линейных алгоритма считаются эквивалентными. В некоторых ситуациях константа в сложности линейного алгоритма может оказаться настолько большой, что даже экспоненциальный алгоритм с маленькой константой на практике будет более эффективным. Эта критика O-обозначения вполне справедлива. Просто следует помнить о том, что она представляет собой эмпирическое правило; алгоритм с оптимальной сложностью не всегда является лучшим.

В табл. 2.2 перечислены все категории сложности с указанием определенного количества элементов, чтобы читатели могли оценить быстродействие алгоритмов. Как видим, при небольшом количестве элементов продолжительность выполнения алгоритмов изменяется мало. В этом случае постоянные множители, скрытые О-обозначением, могут оказывать большое влияние. Однако чем больше элементов поступает на вход алгоритма, тем сильнее проявляется разница между временем их выполнения. При этом роль постоянных множителей ослабляется. При оценке сложности следует думать "масштабно".

Некоторые оценки сложности в стандарте языка C++ называются *амортизированными*. Это означает, что операции *в среднем* выполняются так, как описано. Однако отдельная операция может выполняться дольше, чем указано. Например, если вы добавляете элементы в динамический массив, то продолжительность выполнения этой операции зависит от того, есть ли в массиве достаточно памяти для еще одного элемента. Если дополнительная память есть, сложность является константной, потому что вставка нового последнего элемента в массив всегда занимает одно и то же время. Однако если памяти недостаточно, то сложность является линейной, потому что в зависимости от количества элементов вам придется выделить новую память и скопировать все элементы. Повторное выделение памяти происходит довольно редко, поэтому любая достаточно длинная последовательность таких операций выполняется так, будто сложность каждой операции является константной. Таким образом, сложность вставки выполняется за амортизированное постоянное время.

Таблица 2.2. Продолжительность выполнения алгоритма в зависимости от его сложности и количества элементов

Сложность	Обозначение	1	2	5	10	50	100	1,000	10,000
Константная	O(1)	1	1	1	1	1	1	1	1
Логарифмическая	O($log(n)$)	1	2	3	4	6	7	10	13
Линейная	O(n)	1	2	5	10	50	100	1,000	10,000
n-log-n	O($nlog(n)$)	1	4	15	40	300	700	10,000	130,000
Квадратичная	$O(n^2)$	1	4	25	100	2,500	10,000	1,000,000	100,000,000

Глава 3

Новые средства языка

Ядро языка C++ и его библиотеку обычно стандартизировали параллельно. Благодаря этому библиотека могла использовать преимущества, предоставляемые новыми средствами языка, а язык мог улучшаться за счет опыта реализации библиотеки. По этой причине стандартная библиотека языка C++ всегда использует конкретные средства языка, которых могло не быть в предыдущих версиях стандарта.

Таким образом, язык C++11 отличается от языка C++98/C++03, а тот, в свою очередь, отличается от языка C++, существовавшего до начала стандартизации. Если не знать об этой эволюции, то можно удивиться новым языковым средствам, используемым в библиотеке. В настоящей главе дан краткий обзор новых средств языка C++11, играющих важную роль в проектировании, понимании и применении стандартной библиотеки C++11. В конце главы описываются некоторые средства, существовавшие до появления стандарта C++11, но не получившие широкой известности.

Когда я писал эту книгу (2010-2011 гг.), не все компиляторы поддерживали новые языковые средства стандарта C++11. В ближайшее время это положение, по-видимому, изменится, поскольку все основные поставщики компиляторов языка C++ принимали участие в процессе его стандартизации[1]. Но некоторое время пользователи будут ограничены в выборе языковых средств, используемых в библиотеке. На протяжении книги все типичные и важные ограничения будут указываться в сносках.

3.1. Новые языковые средства стандарта C++11

3.1.1. Небольшие, но важные синтаксические уточнения

Во-первых, следует упомянуть о двух новых средствах языка C++11, которые являются небольшими, но важными для повседневного программирования.

Пробелы в выражениях с шаблонами

Требование помещать пробел между двумя закрывающими угловыми скобками в шаблонных выражениях отменено:

```
vector<list<int> >; // OK в каждой версии C++
vector<list<int>>;  // OK, начиная с версии C++11
```

В книге (и реальных программах) можно встретить обе эти формы.

---

[1] С текущим состоянием дел можно ознакомиться, например, по адресу `http://wiki.apache.org/stdcxx/C++0xCompilerSupport`. — *Примеч. ред.*

Ключевое слово nullptr и тип std::nullptr_t

Стандарт C++11 позволяет использовать ключевое слово `nullptr` вместо `0` или `NULL`, чтобы отметить тот факт, что указатель не ссылается ни на один объект (в отличие от ситуации, когда указатель не имеет определенного значения). Это новое средство языка позволяет избежать ошибок, возникающих, когда нулевой указатель интерпретируется как целочисленное значение. Например:

```
void f(int);

void f(void*);

f(0);       // вызов f(int)

f(NULL);    // вызов f(int), если NULL == 0,
            // и неоднозначность в противном случае

f(nullptr); // вызов f(void*)
```

Слово `nullptr` — это новое ключевое слово. Константа, заданная с помощью ключевого слова `nullptr`, автоматически конвертируется в любой тип указателя, но не в целочисленный тип. Она имеет тип `std::nullptr_t`, определенный в заголовке `<cstddef>` (см. раздел 5.8.1), так что теперь есть возможность даже перегружать операторы для ситуаций, когда им передается нулевой указатель. Отметим, что тип `std::nullptr_t` относится к элементарным типам данных (см. раздел 5.4.2).

3.1.2. Автоматическое выведение типа с помощью ключевого слова auto

В языке C++11 можно объявить переменную или объект без указания их конкретного типа, используя ключевое слово `auto`.[2] Рассмотрим пример:

```
auto i = 42;  // Переменная i имеет тип int
double f();
auto d = f(); // Переменная d имеет тип double
```

Тип переменной, объявленной с помощью ключевого слова `auto`, выводится из ее инициализатора. Например, в следующем коде требуется инициализатор:

```
auto i; // ОШИБКА: невозможно вывести тип переменной i
```

Разрешается использование дополнительных квалификаторов. Например:

```
static auto vat = 0.19;
```

Использование ключевого слова `auto` особенно полезно в ситуациях, когда тип является очень длинным, а выражение сложным. Например:

---

[2] Отметим, что `auto` — старое ключевое слово, унаследованное от языка C. В качестве противоположности ключевому слову `static` оно означало, что переменная является локальной, но никогда не использовалось, потому что по умолчанию все, что не объявлено как `static`, неявно считалось объявленным как `auto`.

```
vector<string> v;
...
auto pos = v.begin(); // переменная pos имеет тип
                      // vector<string>::iterator
auto l = [] (int x) -> bool { // Переменная l имеет тип лямбда-функции,
             ...,             // принимающей int и возвращающей bool
         };
```

Переменная l является объектом, который представляет собой лямбда-функцию (лямбда-функции рассматриваются в разделе 3.1.10).

3.1.3. Универсальная инициализация и списки инициализации

До появления стандарта C++11 программисты, особенно начинающие, легко могли запутаться в вопросах инициализации переменной или объекта. Инициализация могла осуществляться с помощью круглых или фигурных скобок, а также операторов присваивания.

По этой причине в стандарт C++11 включена концепция универсальной инициализации, означающая, что любая инициализация осуществляется с помощью единообразного синтаксиса. Эта конструкция использует фигурные скобки. Например:

```
int values[] { 1, 2, 3 };
std::vector<int> v { 2, 3, 5, 7, 11, 13, 17 };
std::vector<std::string> cities {
    "Berlin", "New York", "London", "Braunschweig", "Cairo", "Cologne"
};
std::complex<double> c{4.0,3.0}; // эквивалентно to c(4.0,3.0)
```

Список инициализации осуществляет так называемую *инициализацию значениями* (*value initialization*), подразумевающую, что каждая, даже локальная переменная элементарного типа, которая обычно имеет неопределенное начальное значение, инициализируется нулем (или константой `nullptr`, если переменная является указателем):

```
int i;     // Переменная i имеет неопределенное значение
int j{};   // Переменная j инициализируется нулем
int* p;    // Переменная p имеет неопределенное значение
int* q{};  // Переменная q инициализируется константой nullptr
```

Однако *сужающие* инициализации, т.е. уменьшающие точность или модифицирующие передаваемое значение, в фигурных скобка запрещены. Например:

```
int x1(5.3);      // OK, но x1 становится равным 5
int x2 = 5.3;     // OK, но x2 становится равным 5
int x3{5.0};      // ОШИБКА: сужение
int x4 = {5.3};   // ОШИБКА: сужение
char c1{7};       // OK: несмотря на то что 7 - целое значение,
                  // это не сужение
char c2{99999};   // ОШИБКА: сужение (если 99999 выходит
                  // за диапазон char)
std::vector<int> v1 { 1, 2, 4, 5 };     // OK
std::vector<int> v2 { 1, 2.3, 4, 5.6 }; // ОШИБКА: сужение double в int
```

Легко видеть, что для проверки сужения могут рассматриваться даже текущие значения, доступные на этапе компиляции. Как написал Бьярне Страуструп (Bjarne Stroustrup) в документе [*Stroustrup:FAQ*] об этом примере: "Способ, которым стандарт C++11 позволяет избежать множества несогласованностей из-за сужающих преобразований, основан на анализе, когда это возможно, реальных значений (как число 7 в предыдущем примере), а не только типов этих значений. Если значение представляет целевой тип точно, то преобразование не является сужающим. Обратите внимание на то, что преобразование числа с плавающей точкой в целое всегда считается сужающим — даже преобразование 7.0 а 7".

Для поддержки концепции списков инициализации для пользовательских типов стандарт C++11 предусматривает класс `std::initializer_list<>`. Его можно использовать для инициализации с помощью списка значений или в любом другом месте, где требуется список значений. Например:

```
void print (std::initializer_list<int> vals)
{
   for (auto p=vals.begin(); p!=vals.end(); ++p)
   {    //обработка списка значений
        std::cout << *p << "\n";
   }
}

print ({12,3,5,7,11,13,17}); // передача списка значений функции print()
```

При наличии конструкторов как с конкретным количеством аргументов, так и со списком инициализации предпочтение отдается версии со списком инициализации.

```
class P
{
   public:
    P(int,int);
    P(std::initializer_list<int>);
};

P p(77,5);    // вызов P::P(int,int)
P q{77,5};    // вызов P::P(initializer_list)
P r{77,5,42}; // вызов P::P(initializer_list)
P s = {77,5}; // вызов P::P(initializer_list)
```

Если бы конструктора со списком инициализации не было, то для инициализации объектов `q` и `s` был бы вызван конструктор, получающий два аргумента типа `int`, а инициализация объекта `r` была бы некорректной.

Благодаря спискам инициализации теперь ключевое слово `explicit` относится и к конструкторам, принимающим больше одного аргумента. Следовательно, теперь можно запретить автоматические преобразования типов нескольких значений, которые использовались также при инициализации с помощью синтаксиса присваивания.

```
class P
{
   public:
     P(int a, int b) {
     ...
     }
```

```
    explicit P(int a, int b, int c) {
    ...
    }
};

P x(77,5);          // OK
P y{77,5};          // OK
P z {77,5,42};      // OK
P v = {77,5};       // OK (неявное преобразование типа допускается)
P w = {77,5,42};    // ОШИБКА из-за ключевого слова explicit
                    // (неявное преобразование типа не допускается)
void fp(const P&);

fp({47,11});        // OK, неявное преобразование {47,11} в P
fp({47,11,3});      // ОШИБКА из-за ключевого слова explicit
fp(P{47,11});       // OK, явное преобразование {47,11} в P
fp(P{47,11,3});     // OK, явное преобразование {47,11,3} в P
```

Аналогично в конструкторах с ключевым словом `explicit`, получающих список инициализации, запрещены неявные преобразования списков инициализации — как пустых, так и содержащих одно или несколько значений.

3.1.4. Диапазонные циклы for

Стандарт C++11 ввел новую форму цикла `for`, который перебирает все элементы в заданном диапазоне, массиве или коллекции. В других языках программирования такая форма цикла называется *foreach*. Общая синтаксическая конструкция этого цикла имеет следующий вид:

```
for ( decl : coll ) {
    операторы
}
```

где *decl* — объявление каждого элемента перебираемой коллекции *coll*, и к каждому элементу применяются указанные операторы. Например, следующий цикл применяет к каждому значению передаваемого списка инициализации оператор, выводящий это значение в стандартный поток вывода `cout`:

```
for ( int i : { 2, 3, 5, 7, 9, 13, 17, 19 } ) {
    std::cout << i << std::endl;
}
```

Для умножения каждого элемента `elem` вектора `vec` на 3 можно написать следующий код:

```
std::vector<double> vec;
...
for ( auto& elem : vec ) {
   elem *= 3;
}
```

Здесь важно подчеркнуть, что переменная `elem` объявлена как ссылка, потому что в противном случае операторы в теле цикла `for` применяются к локальным копиям элементов в векторе (что иногда бывает полезным).

Это значит, что, для того чтобы избежать вызова копирующего конструктора и деструктора для каждого элемента, необходимо объявить текущий элемент как константную ссылку. Таким образом, обобщенная функция для вывода всех элементов коллекции может быть реализована следующим образом:

```
template <typename T>
void printElements (const T& coll)
{
   for (const auto& elem : coll) {
       std::cout << elem << std::endl;
   }
}
```

Здесь диапазонный цикл `for` эквивалентен следующей конструкции:

```
{
   for (auto _pos=coll.begin(); _pos != coll.end(); ++_pos ) {
       const auto& elem = *_pos;
       std::cout << elem << std::endl;
   }
}
```

В общем случае диапазонный цикл `for`, объявленный как

```
for ( decl : coll ) {
   операторы
}
```

эквивалентен следующей конструкции (если объект *coll* имеет члены `begin()` и `end()`):

```
{
   for (auto _pos=coll.begin(), _end=coll.end(); _pos!=_end; ++_pos ) {
        decl = *_pos;
        операторы
   }
}
```

или, если таких членов нет, следующей конструкции, в которой используются глобальные функции `begin()` и `end()`, принимающие объект *coll* как аргумент:

```
{
   for (auto _pos=begin(coll), _end=end(coll); _pos!=_end; ++_pos ) {
   decl = *_pos;
   операторы
   }
}
```

В результате появляется возможность использовать диапазонный цикл `for` даже в случае списка инициализации, поскольку шаблонный класс `std::initializer_list<>` содержит члены `begin()` и `end()`.

Кроме того, существует правило, позволяющее использовать массивы в стиле языка C, имеющие известный размер. Например, программа

```
int array[] = { 1, 2, 3, 4, 5 };

long sum=0; // накапливает сумму всех элементов
for (int x : array) {
   sum += x;
}

for (auto elem : { sum, sum*2, sum*4 } ) { // выводим на экран
                                           // несколько произведений суммы
                                           // на число
   std::cout << elem << std::endl;
}
```

выводит следующие результаты:

```
15
30
60
```

Обратите внимание на то, что при инициализации элементов с помощью конструкции *decl* в цикле `for` явное преобразование типа становится невозможным. Таким образом, следующий код скомпилирован не будет:

```
class C
{
   public:
   explicit C(const std::string& s); // явное(!) преобразование типа
                                     // из класса string
   ...
};

std::vector<std::string> vs;
for (const C& elem : vs) { // ОШИБКА, преобразование класса string
                           // в класс C не определено
     std::cout << elem << std::endl;
}
```

3.1.5. Семантика перемещения и rvalue-ссылки

Одним из наиболее важных новшеств в стандарте C++11 является поддержка семантики перемещения. Это средство позволяет еще больше приблизиться к основной цели языка C++ и предотвратить создание ненужных копий и временных объектов.

Новое средство является настолько сложным, что, пытаясь дать краткое описание предмета, я все же рекомендую прочитать более подробное введение в эту тему[3].

Рассмотрим следующий пример:

[3] Это введение основано на работах [*Abrahams:RValues*] (с любезного разрешения Дэйва Абрамса), [*Becker:RValues*] (с любезного разрешения Томаса Беккера) и нескольких электронных письмах из переписки с Даниэлем Крюглером, Дитмаром Кюлем и Йенсом Маурером.

```
void createAndInsert (std::multiset<X>& coll)
{
   X x;              // создаем объект типа X
   ...
   coll.insert(x);  // вставляем его в переданную коллекцию
}
```

Здесь мы вставляем в коллекцию новый объект, имеющий функцию-член, которая создает внутреннюю копию передаваемого элемента.

```
namespace std {
   template <typename T, ...> class multiset {
      public:
        ... insert (const T& v); // копия объекта v

        ...
      };
}
```

Это полезно, поскольку коллекция имеет семантику значений и предусматривает возможность вставлять временные объекты или объекты, которые используются или модифицируются после вставки.

```
X x;


coll.insert(x); // вставляет копию объекта x


...
coll.insert(x+x); // вставляет копию временного rvalue-значения
...
coll.insert(x);   // вставляет копию объекта x
                  // (хотя объект x больше не используется)
```

Однако для двух последних вставок было бы хорошо, чтобы передаваемые значения (x+x и x) больше не использовались вызывающим объектом `coll` и он мог избежать создания их копий и вместо этого каким-то образом *переместить* их содержимое в новые элементы. Это особенно полезно, если копирование объекта x связано с большими затратами ресурсов — например, если он представляет собой крупную коллекцию строк. В этом случае быстродействие программы значительно повысилось бы.

В языке C++11 это стало реальностью. Однако программист обязан указать, что перемещение возможно, иначе будут использоваться временные копии объектов. Несмотря на то что в тривиальных случаях компилятор может самостоятельно решать такие задачи, новый механизм позволяет программисту применять это средство в любых ситуациях, где это требует логика. Предыдущий пример можно изменить следующим образом:

```
X x;
coll.insert(x);           // вставляет копию x
                          // (OK, x по-прежнему используется)
...
coll.insert(x+x);         // перемещает (или копирует) содержимое
                          // временного rvalue-значения
```

```
...
coll.insert(std::move(x)); // перемещает (или копирует) содержимое x
                           // в объект coll
```

С помощью функции `std::move()`, объявленной в заголовке `<utility>`, объект `x` можно *переместить*, а не копировать. Однако сама функция `std::move()` не выполняет перемещение, она просто преобразовывает свой аргумент в так называемую rvalue-*ссылку* (rvalue reference), представляющую собой тип, объявленный с двумя амперсандами: `X&&`. Этот новый тип представляет rvalue-значения (анонимные временные объекты, которые могут появляться только в правой части оператора присваивания), допускающие модификацию. По соглашению объект `x` в этом случае считается временным объектом, который уже не нужен, поэтому его содержимое и/или ресурсы можно захватить.

Теперь коллекция может использовать перегруженные версии функции `insert()`, работающий с rvalue-ссылками.

```
namespace std {
  template <typename T, ...> class multiset {
    public:
      ... insert (const T& x); // для lvalue-значений: копирует значение
      ... insert (T&& x);      // for rvalue-значений: перемещает значение
      ...
  };
}
```

Версия для rvalue-значений теперь может быть оптимизирована, чтобы ее реализация *захватывала* (steals) содержимое объекта `x`. Однако для этого нужна помощь со стороны типа объекта `x`, поскольку только члены объекта `x` имеют доступ к его внутренним компонентам. Так, например, для инициализации вставленного элемента можно использовать внутренние массивы и указатели из объекта `x`. Если класс объекта `x` является сложным, это может значительно повысить быстродействие программы. В противном случае нам пришлось бы копировать элемент за элементом. Для инициализации нового внутреннего элемента мы просто вызываем так называемый перемещающий конструктор из класса `X`, который захватывает значение переданного аргумента для инициализации нового объекта. Все сложные типы должны — как это сделано в стандартной библиотеке C++ — иметь такой специальный конструктор, осуществляющий перемещение содержимого существующего элемента в новый элемент.

```
class X {
  public:
    X (const X& lvalue); // копирующий конструктор
    X (X&& rvalue);      // перемещающий конструктор
    ...
};
```

Например, перемещающий конструктор для строк обычно просто присваивает существующий внутренний массив символов новому объекту, а не создает новый массив и не копирует все элементы. То же самое относится и ко всем классам коллекций: вместо создания копий всех элементов мы просто присваиваем внутреннюю память новому объекту. Если в классе нет перемещающего конструктора, используется копирующий конструктор.

Кроме того, следует сделать так, чтобы любая модификация — особенно уничтожение — передаваемого объекта, содержимое которого было *захвачено*, не оказывало

влияния на состояние нового объекта, который стал владельцем значения. Для этого обычно достаточно стереть содержимое передаваемого аргумента (например, присвоив константу `nullptr` его внутреннему члену, ссылающемуся на его элементы).

Стирание содержимого объекта, к которому применяется семантика перемещения, вообще говоря, не является необходимым, но если этого не сделать, то весь механизм становится практически бесполезным. Все классы стандартной библиотеки C++ гарантируют, что после перемещения объект пребывает в *корректном, но неопределенном состоянии.* Иначе говоря, вы можете впоследствии присваивать ему новые значения, но его текущее состояние не определено. Контейнеры из библиотеки STL гарантируют, что после перемещения значений они остаются пустыми.

Точно так же любой нетривиальный класс должен содержать оператор копирующего присваивания и оператор перемещающего присваивания.

```
class X {
   public:
     X& operator= (const X& lvalue);  // оператор копирующего присваивания
     X& operator= (X&& rvalue);       // оператор перемещающего присваивания
     ...
};
```

Для строк и коллекций эти операторы можно было бы реализовать так, чтобы они просто меняли местами внутреннее содержимое и ресурсы. Однако в этом случае также следует стереть содержимое объекта `*this`, потому что он может владеть ресурсами, например блокировками, которые необходимо как можно быстрее освободить. Как и в предыдущих случаях, с формальной точки зрения семантика перемещения не требует этого, но, например, контейнерные классы в стандартной библиотеке C++ делают это в обязательном порядке.

В заключение сделаем два замечания о новых средствах языка: 1) правила перегрузки для rvalue- и lvalue-ссылок и 2) возврат rvalue-ссылок.

Правила перегрузки для rvalue- и lvalue-ссылок

Правила перегрузки для rvalue- и lvalue-значений формулируются следующим образом[4].

- Если реализуется только функция

  ```
  void foo(X&);
  ```

  без `foo(X&&)`, то ее поведение следует стандарту C++98: функцию `foo()` можно вызывать для lvalue-значений, но не для rvalue-значений.

- Если реализуется функция

  ```
  void foo(const X&);
  ```

  без функции `void foo(X&&)`, то ее поведение следует стандарту C++98: функцию `foo()` можно вызывать как для rvalue-значений, так и для lvalue-значений.

- Если реализуются функции

  ```
  void foo(X&);
  void foo(X&&);
  ```

[4] Данная формулировка принадлежит Томасу Беккеру.

или

```
void foo(const X&);
void foo(X&&);
```

то rvalue- и lvalue-значения используются по-разному. Версия функции для rvalue-значений может и обязана использовать семантику перемещения. Таким образом, она может *захватывать* внутреннее состояние и ресурсы передаваемого аргумента.

- Если реализуется функция

```
void foo(X&&);
```

но ни `void foo(X&)`, ни `void foo(const X&)` не реализованы, то функция `foo()` может вызываться для rvalues-значений, но попытка вызвать ее для lvalue-значения вызовет ошибку компиляции. Таким образом, здесь предусматривается только семантика перемещения. Эта возможность используется в библиотеке: например, уникальными указателями (см. раздел 5.2.5), файловыми потоками (см. раздел 15.9.2) или строковыми потоками (см. раздел 15.10.2).

Это означает, что если класс не предусматривает семантику перемещения и содержит только обычный копирующий конструктор и оператор копирующего присваивания, то он может использоваться только для rvalue-ссылок. Таким образом, функция `std::move()` реализует семантику перемещения, если она предусмотрена в классе, или семантику копирования в противном случае.

Возврат rvalue-ссылок

Функцию `move()` не обязательно применять к возвращаемым значениям. В соответствии с правилами языка в стандарте определено, что для кода [5]

```
X foo ()
{
X x;
...
return x;
}
```

гарантируется следующее поведение:

- Если класс `X` имеет доступный копирующий конструктор или конструктор перемещения, компилятор может игнорировать копию. Это так называемая (*именованная*) *оптимизация возвращаемого значения* ((named) return value optimization — (N) RVO). Она была определена еще до появления стандарта C++11 и поддерживалась большинством компиляторов.
- В противном случае, если в классе `X` есть перемещающий конструктор, объект `x` перемещается.

[5] Эта формулировка принадлежит Дэйву Абрамсу.

- В противном случае, если в классе X есть копирующий конструктор, объект x копируется.
- В противном случае возникает ошибка компиляции.

Отметим, что возврат rvalue-ссылки является ошибкой, если возвращаемый объект является локальным и нестатическим.

```
X&& foo ()
{
   X x;
   ...
   return x; // ОШИБКА: возвращает ссылку на несуществующий объект
}
```

Rvalue-ссылка — это разновидность ссылки, и ее возвращение в момент, когда она ссылается на локальный объект, означает, что вы возвращаете ссылку на объект, которого больше нет. Применение функции `std::move()` на это никак не влияет.

3.1.6. Новые строковые литералы

Начиная с версии C++11 можно определять литералы как для неформатированных строк, так и для многобайтовых строк или строк, состоящих из широких символов.

Литералы для неформатированных строк

Такие неформатированные строки позволяют определить последовательность символов, просто записывая ее содержимое в виде набора символов. Это позволяет избавиться от множества управляющих символов, используемых для маскировки специальных символов.

Неформатированная строка начинается с символов `R" (` и заканчивается символами `) "`. Такая строка может включать переход на новую строку. Например, обычный строковый литерал, содержащий две обратных косых черты и символ `n`, можно определить как обычный строковый литерал

```
"\\\\n"
```

или как неформатированный строковый литерал

```
R"(\\n)"
```

Для вставки символов `)"` в неформатированную строку можно использовать разделитель. Таким образом, полный синтаксис неформатированной строки выглядит как `R"`*delim*`(...)`*delim*`"`, где *delim* — последовательность, состоящая из не более чем 16 основных символов, исключая обратную косую черту, пробелы и скобки.

Например, неформатированный строковый литерал

```
R"nc(a\
     b\nc()"
     )nc";
```

эквивалентен следующему обычному строковому литералу:

```
"a\\\n      b\\nc()\"\n      "
```

Следовательно, эта строка содержит символ a, обратную косую черту, символ перехода на новую строку, несколько пробелов, символ b, обратную косую черту, символ n, символ c, символ двойной кавычки, символ перехода на новую строку и несколько пробелов.

Неформатированные строковые литералы особенно полезны при определении регулярных выражений. (Детали изложены в главе 14.)

Закодированные строковые литералы

С помощью *префикса кодировки* (encoding prefix) можно указать конкретное кодирование символов в строковых литералах. В стандарте определены следующие префиксы кодировки.

- Префикс u8 определяет кодировку UTF-8. Строковый литерал в кодировке UTF-8 инициализируется заданными символами в соответствии с кодировкой UTF-8. Эти символы должны иметь тип `const char`.
- Префикс u определяет строковый литерал, состоящий из символов, имеющих тип `char16_t`.
- Префикс U определяет строковый литерал, состоящий из символов, имеющих тип `char32_t`.
- Префикс L определяет строковый литерал, состоящий из символов, имеющих тип `wchar_t`.

Например::

```
L"hello" // определяет строку "hello" как строковый литерал типа wchar_t
```

Перед начальным символом R в неформатированной строке может стоять префикс кодировки.

Детали использования разных кодировок для интернационализации описаны в главе 16.

3.1.7. Ключевое слово noexcept

Стандарт C++11 предусматривает ключевое слово `noexcept`. Его можно использовать для того, чтобы указать, что функция не может генерировать исключения или не предназначена для этого. Например, код

```
void foo () noexcept;
```

объявляет, что функция `foo()` не может генерировать исключения. Если исключение не было обработано локально в функции `foo()`, т.е. если функция `foo()` все же сгенерировала исключение, программа прекращает работу, вызывая функцию `std::terminate()`, которая по умолчанию вызывает функцию `std::abort()` (см. раздел 5.8.2).

Ключевое слово `noexcept` решает множество проблем, связанных со спецификациями исключений. Процитируем работу [*N3051:DeprExcSpec*] (с любезного разрешения Дуга Грегора).

- Проверка во время выполнения программы. Спецификации исключения в языке C++ проверяются во время выполнения программы, а не на этапе компиляции, поэтому у программиста нет гарантии, что все исключения будут обработаны. Сбой во время выполнения (приводящий к вызову функции `std::unexpected()`) сам по себе не подлежит восстановлению.
- Излишние затраты ресурсов при выполнении программы. Проверка на этапе выполнения программы требует, чтобы компилятор сгенерировал дополнительный код, который мешает оптимизации.
- Недопустимость в обобщенном коде. В обобщенном коде обычно невозможно знать заранее, какие типы исключений могут генерироваться операторами, принадлежащими шаблонным аргументам, поэтому невозможно написать точную спецификацию исключения.

На практике оказались полезными только две спецификации, связанные с генерацией исключений: оператор может генерировать любое исключение или не должен генерировать исключения вовсе. Первое требование выражается в игнорировании спецификации исключений вообще, а второе можно выразить с помощью оператора `throw()`, хотя это редко делают из-за снижения быстродействия программы.

Ключевое слово `noexcept` не требует разворачивания стека, поэтому программисты теперь могут явно гарантировать отсутствие исключений без дополнительных затрат. В результате использование спецификаций исключений из стандарта C++11 исключено.

Можно даже указать условие, при котором функция не генерирует никаких исключений. Например, для любого типа *Type* глобальная функция `swap()` обычно определяется следующим образом:

```
void swap (Type& x, Type& y) noexcept(noexcept(x.swap(y)))
{
    x.swap(y);
}
```

Здесь в спецификаторе `noxcept(...)` можно задать булево выражение, при котором функция не может генерировать исключение. Если в спецификаторе `noexcept` не указано никаких условий, подразумевается условие `noexcept(true)`.

В данном случае условие имеет вид `noexcept(x.swap(y))`. Здесь используется *оператор* `noexcept`, который возвращает значение `true`, если вычисляемое выражение, заданное в скобках, не может генерировать исключения. Таким образом, глобальная функция `swap()` указывает, что она не генерирует никаких исключений, если функция-член `swap()`, вызванная для первого аргумента, не генерирует исключений.

Другой пример связан с оператором перемещающего присваивания, который применяется к паре значений следующим образом:

```
pair& operator= (pair&& p)
        noexcept(is_nothrow_move_assignable<T1>::value &&
                 is_nothrow_move_assignable<T2>::value);
```

Здесь используется свойство типа `is_nothrow_move_assignable`, которое проверяет, можно ли применять к передаваемому типу оператор перемещающего присваивания, который не может генерировать исключения (см. раздел 5.4.2).

В соответствии с документом [*N3279:LibNoexcept*], ключевое слово `noexcept` включено в библиотеку для реализации консервативного подхода (слова и фразы, *набранные курсивом*, цитируются буквально).

- *Каждая библиотечная функция... которая ...не может генерировать исключения* и не задает никакого непредсказуемого поведения — например, вследствие нарушения предусловия — *должна быть помечена безусловной спецификацией* `noexcept`.
- *Если можно доказать, что библиотечная функция обмена, перемещающий конструктор или оператор перемещающего присваивания... не могут генерировать исключение путем применения оператора* `noexcept`, *они должны быть помечены условной спецификацией* `noexcept`. *Ни одна другая функция не должна использовать условную спецификацию* `noexcept`.
- *Ни один библиотечный деструктор не должен генерировать исключения. Он должен использовать неявно указанную спецификацию, гарантирующую отсутствие генерации исключений.*
- *Библиотечные функции, разработанные с учетом совместимости с языком C... могут быть помечены безусловной спецификацией* `noexcept`.

Отметим, что спецификация `noexcept` преднамеренно не применяется к функциям языка C++, имеющим предусловие, нарушение которого может привести к непредсказуемому поведению. Это позволяет обеспечить "безопасный режим" реализации библиотек путем генерирования исключения "нарушение предусловия" в случае неправильного использования функции.

На протяжении книги я обычно пропускаю спецификации `noexcept`, чтобы не загромождать изложение.

3.1.8. Ключевое слово `constexpr`

Начиная с версии C++11 с помощью ключевого слова `constexpr` можно отмечать выражения, вычисляемые на этапе компиляции. Например:

```
constexpr int square (int x)
{
   return x * x;
}
float a[square(9)]; // OK в версии C++11:
                    // массив a содержит 81 элемент
```

Это ключевое слово решает проблему, возникшую в стандарте C++98 при использовании предельных числовых значений (см. раздел 5.3). До стандарта C++11 такое выражение, как

```
std::numeric_limits<short>::max()
```

не могло использоваться как целочисленная константа, хотя и было функционально эквивалентным макросу `INT_MAX`. С появлением стандарта C++11 такие выражения объявляются с ключевым словом `constexpr`. Например, его можно использовать для объявления массива и вычислений на этапе компиляции (метапрограммирования):

```
std::array<float,std::numeric_limits<short>::max()> a;
```

На протяжении книги я обычно пропускаю спецификации constexpr, чтобы не загромождать изложение.

3.1.9. Новые возможности шаблонов

Вариативные шаблоны

В стандарте C++11 шаблоны могут иметь параметры, принимающие переменное количество шаблонных аргументов. Эта возможность называется *вариативными шаблонами* (variadic templates). В следующем примере их можно использовать для вызова функции print() с аргументами разных типов.

```
void print ()
{
}

template <typename T, typename... Types>
void print (const T& firstArg, const Types&... args)
{
   std::cout << firstArg << std::endl; // выводим на экран первый аргумент
   print(args...);                     // вызываем print() для
// остальных аргументов
}
```

При передаче одного или нескольких аргументов используется шаблонная функция, в которой отдельное определение первого аргумента позволяет выводить его на экран, а затем рекурсивно вызывать функцию print() с оставшимися аргументами. Для завершения рекурсии предоставляется нешаблонная перегруженная версия функции print().

Например, вызов функции

```
print (7.5, "hello", std::bitset<16>(377), 42);
```

приводит к следующему результату (см. раздел 12.5.1, в котором подробно описаны битовые множества):

```
7.5
hello
0000000101111001
42
```

Отметим, что корректность этого примера в настоящее время также является предметом обсуждения. Причина заключается в том, что вариативная форма с одним аргументом формально совпадает с невариативной формой с одним аргументом; однако компиляторы обычно допускают следующий код:

```
template <typename T>
void print (const T& arg)
{
   std::cout << arg << std::endl;
}
```

```
template <typename T, typename... Types>
void print (const T& firstArg, const Types&... args)
{
   std::cout << firstArg << std::endl; // выводим на экран первый аргумент
   print(args...);                     // вызываем print() для
// остальных аргументов
}
```

В вариативных шаблонах оператор `sizeof...(args)` возвращает количество аргументов.

Эту функциональную возможность интенсивно использует класс `std::tuple<>` (см. раздел 5.1.2).

Шаблонный псевдоним (шаблонный оператор typedef)

В стандарте C++11 поддерживаются определения шаблонных (частичных) типов. Однако, поскольку все подходы, основанные на использовании ключевого слова `typename`, по некоторой причине оказались неработоспособными, было введено ключевое слово `using` и термин *псевдоним шаблона* (alias template). Например, после операторов

```
template <typename T>

using Vec = std::vector<T,MyAlloc<T>>; // стандартный вектор,
                                       // использующий собственный механизм
                                       // распределения памяти
```

выражение

```
Vec<int> coll;
```

становится эквивалентным выражению

```
std::vector<int,MyAlloc<int>> coll;
```

Другой пример можно найти в разделе 5.2.5.

Другие новые шаблонные средства

В стандарте C++11 шаблонные функции (см. раздел 3.2) могут иметь шаблонные аргументы, заданные по умолчанию. Кроме того, локальные типы теперь можно использовать как шаблонные аргументы, а функции с внутренним связыванием теперь можно использовать в качестве аргументов для нетипизированных шаблонов указателей или ссылок на функции.

3.1.10. Лямбда-выражения и лямбда-функции

В стандарте C++11 появились *лямбда-выражения* и *лямбда-функции*, позволяющие создавать определения подставляемых функций, которые можно использовать в качестве параметра или локального объекта.

Лямбда-выражения и лямбда-функции изменяют способ использования стандартной библиотеки C++. Например, в разделах 6.9 и 10.3 показано, как использовать лямбда-выражения и лямбда-функции с алгоритмами и контейнерами из библиотеки STL. В разделе 18.1.2 продемонстрировано, как с помощью лямбда-выражений и лямбда-функций определить код, который может работать в параллельном режиме.

Синтаксис лямбда-выражений и лямбда-функций

Лямбда-функция — это описание функциональной возможности, которую можно определить в операторе и выражении. Таким образом, лямбда-функцию можно использовать в качестве подставляемой.

Минимальная лямбда-функция не имеет параметров и просто делает что-то. Например:

```
[] {
   std::cout << "hello lambda" << std::endl;
}
```

Эту функцию можно вызывать непосредственно

```
[] {
   std::cout << "hello lambda" << std::endl;
} (); // выводит на экран "hello lambda"
```

или передавать вызываемым объектам

```
auto l = [] {
   std::cout << "hello lambda" << std::endl;
};
...
l(); // выводим на экран "hello lambda"
```

Как видим, лямбда-функции всегда предшествует так называемый *инициатор лямбда-функции* (lambda introducer): квадратные скобки, внутри которых можно определить так называемый захват для доступа к нестатическим внешним объектам в лямбда-функции. Если доступ к внешним данным не нужен, квадратные скобки остаются пустыми, как в данном случае. В лямбда-функциях можно использовать статические объекты, например `std::cout`.

Между инициатором лямбда-функции и телом лямбда-функции можно указать параметры, ключевое слово `mutable`, спецификацию исключения, спецификаторы атрибутов и тип возвращаемого значения. Все это не обязательно, но если один из перечисленных пунктов присутствует, круглые скобки для параметров становятся обязательными. Таким образом, синтаксис лямбда-функции выглядит либо так:

```
[...] {...}
```

либо так:

```
[...] (...) mutableopt throwSpecopt ->retTypeopt {...}
```

Лямбда-функция может иметь параметры, указанные в скобках, как обычная функция.

```
auto l = [] (const std::string& s) {
            std::cout << s << std::endl;
         };
l("hello lambda"); // выводит на экран "hello lambda"
```

Однако следует подчеркнуть, что лямбда-функции не могут быть шаблонными. В них всегда необходимо указывать все типы.

Лямбда-функция может возвращать какой-нибудь объект. Если тип возвращаемого объекта не указан, он выводится из его значения. Например, следующая функция возвращает значение типа `int`:

```
[] {
   return 42;
}
```

Для указания типа возвращаемого значения можно использовать новую синтаксическую конструкцию языка C++, которая применяется и для обычных функций (см. раздел 3.1.12). Например, следующая лямбда-функция возвращает число `42.0`:

```
[] () -> double {
   return 42;
}
```

В данном случае необходимо указать тип возвращаемого значения после обязательных в данном случае скобок, предназначенных для аргументов, и символов "->."

Между параметрами и указанием типа возвращаемого значения или телом можно также задать спецификацию исключения, как при работе с обычной функцией. Однако для обычных функций спецификации исключения объявлены устаревшими (см. раздел 3.1.7).

Захваты (доступ к внешней области видимости)

В инициаторе лямбда-функции (квадратных скобках перед лямбда-функцией) можно задать *список захвата* (capture) для доступа к данным из внешней области видимости, которые не передаются как аргументы.

- Символы `[=]` означают, что внешняя область видимости передается в лямбда-функцию по значению. Таким образом, можно прочитать, но не модифицировать все данные, которые были доступными для чтения при определении лямбда-функции.
- Символы `[&]` означают, что внешняя область видимости передается в лямбда-функцию по ссылке. Таким образом, при определении лямбда-функции можно изменять данные, которые были корректными в момент определения лямбда-функции, при условии, что их можно изменять в принципе.

Кроме того, для каждого объекта в лямбда-функции необходимо указать режим доступа к нему: по значению или по ссылке. Это позволяет ограничивать доступ и смешивать разные режимы. Например, операторы

```
int x=0;
int y=42;
auto qqq = [x, &y] {
              std::cout << "x: " << x << std::endl;
              std::cout << "y: " << y << std::endl;
```

```
            ++y; // OK
        };
x = y = 77;
qqq();
qqq();
std::cout << "final y: " << y << std::endl;
```

приводят к следующим результатам:

```
x: 0
y: 77
x: 0
y: 78
final y: 79
```

Поскольку объект x передается по значению, его нельзя модифицировать в лямбда-функции; оператор ++x в лямбда-функции не скомпилируется. Поскольку объект y передается по ссылке, его можно модифицировать, поэтому двойной вызов лямбда-функции дважды увеличит присвоенное изначально значение 77.

Вместо списка захвата [x, &y] можно написать список [=, &y] и передать объект y по ссылке, а все остальные объекты — по значению.

Для того чтобы смешать передачу по ссылке и по значению, лямбда-функцию можно объявить с ключевым словом mutable. В этом случае объекты передаются по значению, но в функции-объекте, определенной в лямбда-функции, переданное значение можно изменить. Например, код

```
int id = 0;
auto f = [id] () mutable {
          std::cout << "id: " << id << std::endl;
          ++id; // OK
         };
id = 42;
f();
f();
f();
std::cout << id << std::endl;
```

выводит следующие результаты:

```
id: 0
id: 1
id: 2
42
```

Лямбда-функцию можно сравнить с функциональным объектом (см. раздел 6.10).

```
class {
   private:
     int id; // копирование внешнего id
   public:
     void operator() () {
         std::cout << "id: " << id << std::endl;
```

```
        ++id; // OK
    }
};
```

Благодаря ключевому слову `mutable` оператор вызова функции `operator ()` определен как неконстантная функция-член, т.е. существует возможность изменять значение переменной `id`. Итак, благодаря ключевому слову `mutable` лямбда-функция сохраняет текущее состояние, даже если состояние передается по значению. Без ключевого слова `mutable`, что является обычной ситуацией, оператор вызова функции `operator ()` становится константной функцией-членом, и вы имеете право лишь читать объекты, переданные по значению. Другой пример использования ключевого слова `mutable` с лямбда-функциями приведен в разделе 10.3.2, в котором также освещены возможные проблемы.

Типы лямбда-функций

Типом лямбда-функции является анонимный функциональный объект (или функтор), являющийся уникальным для каждого лямбда-выражения. Таким образом, для объявления объектов такого типа необходимы шаблоны или ключевое слово `auto`. Если необходим тип, можно использовать ключевое слово `decltype` (см. раздел 3.1.11), которое, например, требуется при передаче лямбда-функции в качестве хеш-функции или критерия сортировки для ассоциативных или неупорядоченных контейнеров. Подробности описаны в разделах 6.9 и 7.9.7.

В качестве альтернативы для указания общего типа, предназначенного для функционального программирования, можно использовать шаблонный класс `std::function<>`, предоставляемый стандартной библиотекой языка C++ (см. раздел 5.4.4). Этот шаблонный класс обеспечивает единственный способ задать тип значения, возвращаемого лямбда-функцией:

```
// lang/lambda1.cpp

#include<functional>
#include<iostream>

std::function<int(int,int)> returnLambda ()
{
   return [] (int x, int y) {
              return x*y;
          };
}

int main()
{
   auto lf = returnLambda();
   std::cout << lf(6,7) << std::endl;
}
```

Ниже приведен результат работы этой программы.

```
42
```

3.1.11. Ключевое слово decltype

С помощью ключевого слова `decltype` можно позволить компилятору самому распознать тип выражения. Это реализация часто используемого оператора `typeof`. Поскольку существующие реализации оператора `typeof` противоречивы и неполны, стандарт C++11 ввел новое ключевое слово. Например:

```
std::map<std::string,float> coll;
decltype(coll)::value_type elem;
```

Одно из применений ключевого слова `decltype` — объявление типов возвращаемых значений (см. ниже). Другое применение относится к метапрограммированию (см. раздел 5.4.1) и передаче типа лямбда-функции (см. раздел 10.3.4).

3.1.12. Новый синтаксис объявления функций

Иногда тип значения, возвращаемого функцией, зависит от выражения, имеющего аргументы. Однако код наподобие

```
template <typename T1, typename T2>
decltype(x+y) add(T1 x, T2 y);
```

до появления стандарта C++11 был невозможен, поскольку тип возвращаемого значения в этом случае использует объекты, которые не были объявлены или которые еще не появились в области видимости.

В стандарте C++11 существует альтернативное объявление типа значения, возвращаемого функцией, которое размещается после списка параметров.

```
template <typename T1, typename T2>
auto add(T1 x, T2 y) -> decltype(x+y);
```

Это объявление использует тот же синтаксис, что и лямбда-функции при объявлении типов возвращаемых значений (см. раздел 3.1.10).

3.1.13. Перечисления с ограниченной областью видимости

Стандарт C++11 позволяет определять *перечисления с ограниченной областью видимости* (scoped enumerations), известные также как *строгие перечисления* (strong enumerations) или *классы перечислений* (enumeration classes). Эти перечисления являются более точной реализацией перечисляемых значений (перечислителей) в языке C++.

Например:

```
enum class Salutation : char { mr, ms, co, none };
```

Следует подчеркнуть, что ключевое слово `class` после `enum` является обязательным.

Перечисления с ограниченной областью видимости имеют следующие преимущества.

- Неявные преобразования в тип `int` и из типа `int` невозможны.
- Значения вроде `mr` не являются частью области видимости, в которой объявлено перечисление. Вместо него следует использовать конструкцию `Salutation::mr`.

- Можно явно определить базовый тип (в данном случае `char`) и гарантировать размер элементов перечисления (если пропустить выражение "`: char`", то по умолчанию будет использоваться тип `int`).
- Возможны предваряющие объявления перечислимых типов, исключающие необходимость повторной компиляции модулей с новыми значениями перечислений при условии, что используется только данный тип.

Обратите внимание на то, что свойство типа `std::underlying_type` позволяет определить базовый тип перечисления (см. раздел 5.4.2).

Примерами *перечислений с ограниченной областью видимости* являются условные коды ошибок (см. раздел 4.3.2).

3.1.14. Новые фундаментальные типы данных

В стандарте C++11 определены новые фундаментальные типы данных.

- `char16_t` и `char32_t` (см. раздел 16.1.3)
- `long long` и `unsigned long long`
- `std::nullptr_t` (см. раздел Section 3.1.1)

3.2. Старые "новые" средства языка

Несмотря на то что стандарту C++98 уже больше десяти лет, программисты все еще иногда удивляются некоторым средствам языка. Рассмотрим некоторые из них.

Нетипизированные шаблонные параметры

Помимо типизированных параметров допускаются нетипизированные. Нетипизированный параметр рассматривается как часть типа. Например, стандартному классу `bitset<>` (см. раздел 12.5) можно передать количество битов в виде шаблонного аргумента. Следующие операторы определяют два битовых поля: одно из 32 битов, другое — из 50:

```
bitset<32> flags32; // битовое поле из 32 битов
bitset<50> flags50; // битовое поле из 50 битов
```

Эти битовые множества имеют разные типы, потому что они используют разные шаблонные аргументы. Таким образом, их нельзя присваивать друг другу и сравнивать между собой, не предусмотрев соответствующего преобразования типа.

Шаблонные параметры, заданные по умолчанию

Шаблонные классы могут иметь аргументы, заданные по умолчанию. Например, следующее объявление позволяет объявлять объекты класса `MyClass` с одним или двумя шаблонными аргументами:

```
template <typename T, typename container = vector<T>>
class MyClass;
```

Если передать только один аргумент, то вторым будет использоваться параметр, заданный по умолчанию.

```
MyClass<int> x1; // эквивалент MyClass<int,vector<int>>
```

Отметим, что шаблонные аргументы, заданные по умолчанию, можно определять с помощью предыдущих аргументов.

Ключевое слово typename

Ключевое слово `typename` было введено для того, чтобы указать, что следующий за ним идентификатор обозначает тип. Рассмотрим следующий пример:

```
template <typename T>
class MyClass {
   typename T::SubType * ptr;
   ...
};
```

где ключевое слово `typename` используется для того, чтобы разъяснить, что `SubType` — это тип, определенный в классе `T`. Таким образом, `ptr` — это указатель на объект типа `T::SubType`. Без ключевого слова `typename` имя `SubType` рассматривалось бы как имя статического члена, а значит, выражение

```
T::SubType * ptr
```

означало бы умножение значения объекта `SubType` типа `T` на переменную `ptr`.

Поскольку в программе указано, что `SubType` — это тип, то любой тип, используемый вместо параметра `T`, должен иметь внутренний тип `SubType`. Например, использование типа `Q` в качестве шаблонного аргумента возможно только в том случае, если он содержит определение внутреннего типа `SubType`.

```
class Q {
   typedef int SubType;
   ...
};

MyClass<Q> x; // OK
```

В таком случае переменная `ptr`, являющаяся членом класса `MyClass<Q>`, представляла бы собой указатель на тип `int`. Однако подтип также может быть абстрактным типом данных, таким, как класс:

```
class Q {
   class SubType;
   ...
};
```

Отметим, что ключевое слово `typename` необходимо использовать всегда, если требуется квалифицировать идентификатор шаблона в качестве типа, даже если другая

интерпретация лишена смысла. Итак, общее правило в языке C++ утверждает, что любой идентификатор шаблона рассматривается как значение, если он не квалифицирован ключевым словом `typename`.

Помимо всего прочего, ключевое слово `typename` можно использовать вместо ключевого слова `class` в объявлении шаблона.

```
template <typename T> class MyClass;
```

Шаблонные члены

Функции-члены класса могут быть шаблонными. Однако шаблонные функции-члены не могут быть виртуальными. Например:

```
class MyClass {
   ...
   template <typename T>
   void f(T);
};
```

где выражение `MyClass::f` объявляет набор функций-членов с параметрами любого типа. Вы можете передавать ей любой аргумент, при условии, что его тип предусматривает все операторы, выполняемые в функции `f()`.

Это свойство часто используется для поддержки автоматического преобразования типов членов в шаблонных классах. Например, в следующем определении аргумент x функции `assign()` должен иметь точно такой же тип, как и вызывающий объект:

```
template <typename T>
class MyClass {
   private:
     T value;
   public:
     void assign (const MyClass<T>& x) { // x должен иметь такой же
                                         // тип, что и *this
        value = x.value;
     }
     ...
};
```

При таком объявлении было бы ошибкой использовать разные шаблонные типы для объектов в функции `assign()`, даже если предусмотрено автоматическое преобразование одного типа в другой.

```
void f()
{
   MyClass<double> d;
   MyClass<int> i;

   d.assign(d); // OK
   d.assign(i); // ОШИБКА: i имеет тип MyClass<int>
                // а требуется тип MyClass<double>
}
```

Задав другой шаблонный тип для функции-члена, можно ослабить требование строгого совпадения типов. Шаблонный аргумент функции-члена может иметь любой шаблонный тип. Тогда, поскольку эти типы допускают присваивание, можно написать следующий код:

```
template <typename T>
class MyClass {
   private:
     T value;
   public:
     template <typename X>                  // Шаблонный член
     void assign (const MyClass<X>& x) { // Разрешены другие шаблонные типы
         value = x.getValue();
     }
     T getValue () const {
        return value;
     }
     ...
};

void f()
{
     MyClass<double> d;
     MyClass<int> i;

     d.assign(d); // OK
     d.assign(i); // OK (int можно присваивать переменной типа double)
}
```

Отметим, что аргумент x функции `assign()` теперь отличается от типа объекта `*this`. Таким образом, у нас нет непосредственного доступа к закрытым и защищенным членам класса `MyClass<>`. Вместо этого мы должны использовать что-то вроде функции `getValue()`.

Особым видом шаблонного члена является *шаблонный конструктор* (template constructor). Шаблонные конструкторы обычно используются для осуществления неявных преобразований типа при копировании объектов. Обратите внимание на то, что шаблонный конструктор не подавляет неявное объявление копирующего конструктора. Если типы совпадают точно, генерируется и вызывается неявный копирующий конструктор. Например:

```
template <typename T>
class MyClass {
public:
   // копирующий конструктор с неявным преобразованием типов
   // не подавляет автоматический копирующий конструктор
   template <typename U>
   MyClass (const MyClass<U>& x);
   ...
};

void f()
{
   MyClass<double> xd;
```

```
    ...
    MyClass<double> xd2(xd); // вызывает неявно сгенерированный
                             // копирующий конструктор
    MyClass<int> xi(xd);     // вызывает шаблонный конструктор
    ...
}
```

Здесь тип объекта `xd2` совпадает с типом объекта `xd` и поэтому инициализируется с помощью неявно сгенерированного копирующего конструктора. Тип объекта `xi` отличается от типа объекта `xd` и поэтому инициализируется шаблонным конструктором. Таким образом, если вы реализуете шаблонный конструктор, не забудьте предусмотреть конструктор по умолчанию, если его поведение по умолчанию вас не устраивает. Другой пример шаблонных членов класса приведен в разделе 5.1.1.

Вложенные шаблонные классы

Вложенные классы также могут быть шаблонными.

```
template <typename T>
class MyClass {
    ...
    template <typename T2>
    class NestedClass;
    ...
};
```

3.2.1. Неявная инициализация фундаментальных типов

Если вы используете синтаксис явного вызова конструктора без аргументов, объекты фундаментальных типов инициализируются нулем.

```
int i1;          // неопределенное значение
int i2 = int();  // инициализируется нулем
int i3{};        // инициализируется нулем (по стандарту C++11)
```

Это средство позволяет писать шаблонный код, гарантирующий, что переменные любого типа всегда будут иметь определенное значение, заданное по умолчанию. Например, в следующей функции инициализация гарантирует, что объект `x` фундаментального типа инициализируется нулем:

```
template <typename T>
void f()
{
    T x = T();
    ...
}
```

Если шаблонная функция выполняет принудительную инициализацию нулем, то значение, возвращаемое ею, называется *значением, инициализированным нулем* (zero initialized). В противном случае оно *инициализируется значением по умолчанию.*

3.2.2. Определение функции main()

Следует разъяснить один важный аспект языка, который часто понимают неправильно: корректные и переносимые версии функции main(). В соответствии со стандартом языка C++ переносимыми являются только два определения функции main().

```
int main()
{
   ...
}
```

и

```
int main (int argc, char* argv[])
{
   ...
}
```

Здесь аргумент argv (массив аргументов командной строки) можно также определить как char**. Отметим, что в функции main в обязательном порядке требуется тип возвращаемого значения int.

Функцию main() не обязательно завершать оператором return. В отличие от языка C, в языке C++ в конце функции main() неявно определен оператор

```
return 0;
```

Это значит, что каждая программа, осуществляющая выход из функции main() без выполнения оператора return, считается завершенной успешно. Любое значение, отличающееся от нуля, означает ошибку (предопределенные коды ошибок перечислены в разделе 5.8.2). В связи с этим мои примеры в книге никогда не содержат оператор return в конце функции main().

Для завершения программы на языке C++ без выхода из функции main() обычно следует использовать функции exit(), quick_exit() (начиная со стандарта C++11) или terminate(). Подробности описаны в разделе 5.8.2.

Глава 4

Общие принципы

В этой главе описаны основные принципы стандартной библиотеки C++, необходимые для работы со всеми или с большинством из ее компонентов.

- Пространство имен std.
- Имена и форматы заголовочных файлов.
- Общие принципы обработки ошибок и исключений.
- Вызываемые объекты.
- Основы параллельности и многопоточности.
- Краткое описание распределителей памяти.

4.1. Пространство имен `std`

При использовании разных модулей и/или библиотек всегда существует вероятность конфликтов имен, которые возникают из-за того, что в модулях и библиотеках один и тот же идентификатор может использоваться для названия разных сущностей. В языке C++ эта проблема была решена введением *пространств имен*. Пространство имен — это определенная область видимости идентификаторов. В отличие от классов, пространства имен открыты для расширений, которые могут определяться в любых исходных текстах. Таким образом, в пространстве имен можно определять компоненты, распределенные по нескольким физическим модулям. Типичным примером такого компонента является стандартная библиотека C++, поэтому в ней используется пространство имен.

Практически все идентификаторы стандартной библиотеки C++ определяются в пространстве имен `std`. В стандарте C++11 это также относится к идентификаторам, введенным в документе TR1 и помещенным в пространство имен `std::tr1` (см. раздел 2.1).

Кроме того, в настоящее время в языке C++ есть пространство имен `posix`, которое зарезервировано, но пока не используется в стандартной библиотеке.

В следующем списке перечислены пространства имен, вложенные в пространство имен `std` в стандартной библиотеке C++:

- `std::rel_ops` (см. раздел 5.5.3);
- `std::rel_chrono` (см. раздел 5.7.1);
- `std::placeholders` (см. раздел 6.10.3);
- `std::regex_constants` (см. раздел 14.6);
- `std::this_thread` (см. раздел 18.3.7).

В соответствии с концепцией пространств имен существуют три варианта использования идентификатора из стандартной библиотеки C++.

- Явная квалификация идентификатора. Например, можно написать `std::ostream` вместо `ostream`. Полная команда вывода может иметь следующий вид:

```
std::cout << std::hex << 3.4 << std::endl;
```

- *Объявление using*. Например, следующий фрагмент программы предоставляет локальную возможность пропустить префикс `std::` для объекта `cout` и модификатор формата `endl`:

```
using std::cout;
using std::endl;
```

- В этом случае предыдущий оператор записывается так:

```
cout << std::hex << 3.4 << end!;
```

- *Директива using*. Это простейший вариант. После выполнения директивы `using` для пространства имен `std` все идентификаторы этого пространства доступны так, будто они были объявлены глобально. Например, оператор

```
using namespace std;
```

- позволяет написать

```
cout << hex << 3.4 << endl;
```

- Отметим, что в сложных программах это может привести к случайным конфликтам имен или, что еще хуже, к непредсказуемым последствиям из-за запутанных правил перегрузки. Никогда не используйте директиву `using`, если контекст неясен (например, в заголовочных файлах).

Примеры в книге довольно невелики, поэтому в завершенных примерах программ обычно используются директивы `using`.

4.2. Заголовочные файлы

В процессе стандартизации в язык C++ для всех идентификаторов было внедрено пространство имен `std`. Это изменение не имеет обратной совместимости со старыми заголовочными файлами, в которых идентификаторы стандартной библиотеки C++ объявлялись в глобальной области видимости. Кроме того, в процессе стандартизации изменились некоторые интерфейсы классов (впрочем, при этом по возможности обеспечивалась обратная совместимость). Для этой цели был разработан новый стиль имен стандартных заголовочных файлов, позволяющий разработчикам компиляторов сохранить обратную совместимость со старыми заголовочными файлами.

Определение новых имен для стандартных заголовочных файлов позволило стандартизировать расширения заголовочных файлов. Раньше использовались разные варианты расширений (например, `.h`, `.hpp` и `.hxx`). Однако решение, принятое относительно нового стандартного расширения заголовочных файлов, может показаться неожиданным — теперь стандартные заголовочные файлы вообще не имеют расширений. Таким образом,

директивы include для стандартных заголовочных файлов теперь выглядят примерно так:

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

Аналогичное правило действует для заголовочных файлов в стандарте языка C. Теперь заголовочные файлы языка C снабжаются префиксом c вместо прежнего расширения .h.

```
#include <cstdlib> // было: <stdlib.h>
#include <cstring> // было: <string.h>
```

В этих заголовочных файлах все идентификаторы объявляются в пространстве имен std.

Одно из преимуществ такого стиля именования файлов заключается в том, что он позволяет легко отличить старый заголовочный файл для функций работы со строками char* из языка C от стандартного заголовочного файла C++ для работы с классом string.

```
#include <string>  // Класс string из языка C++
#include <cstring> // Функции char* из языка C
```

Новая схема именования заголовочных файлов не означает, что файлы стандартных заголовков не имеют расширений с точки зрения операционной системы. Выполнение директивы include для стандартных заголовочных файлов зависит от реализации. Системы программирования C++ могут добавлять расширения и даже использовать встроенные объявления, не читая файл. Однако на практике большинство систем просто включает заголовок из файла, имя которого точно совпадает с именем, указанным в директиве include. В большинстве систем *стандартные* заголовочные файлы C++ просто не имеют расширений. В принципе, целесообразно указывать расширения для ваших собственных заголовочных файлов, чтобы упростить их идентификацию в файловой системе.

Для поддержки совместимости с языком C в стандарте сохранена поддержка "старых" стандартных заголовочных файлов языка C. При необходимости можно написать директиву

```
#include <stdlib.h>
```

В этом случае идентификаторы объявляются и в глобальной области видимости, и в пространстве имен std. Фактически эти заголовочные файлы ведут себя так, как если бы все идентификаторы были объявлены в пространстве имен std, после чего была выполнена директива using.

В стандарте нет спецификации заголовочных файлов C++ "старого" формата, таких как <iostream.h>. Таким образом, они не поддерживаются. На практике большинство поставщиков будут поддерживать их для обеспечения обратной совместимости. Отметим, что изменения в заголовках не ограничиваются введением пространства имен std. В общем, следует либо использовать старые имена заголовочных файлов, либо переключиться на новые стандартизированные имена.

4.3. Обработка ошибок и исключений

Стандартная библиотека языка C++ неоднородна. Она содержит программы из разнообразных источников, отличающихся стилями проектирования и реализации. Ярким

примером таких различий является обработка ошибок и исключений. Одни части библиотеки, например строковые классы, поддерживают подробную обработку ошибок, проверяя все возможные проблемы и генерируя исключение, если возникла ошибка. Другие компоненты, такие как стандартная библиотека шаблонов STL и массивы `valarray`, оптимизируются по быстродействию в ущерб безопасности, поэтому они редко проверяют логические ошибки и генерируют исключения только при возникновении ошибок времени выполнения.

4.3.1. Стандартные классы исключений

Все исключения, генерируемые языком или библиотекой, являются производными от базового класса `exception`. Этот класс является корнем иерархии, показанной на рис. 4.1. Стандартные классы исключений можно разделить на три категории.

1. Языковая поддержка.
2. Логические ошибки.
3. Ошибки времени выполнения.

Логических ошибок обычно удается избежать, поскольку их причины, например нарушение предусловия, находятся в самой программе. Ошибки времени выполнения, например нехватка ресурсов, вызываются причинами, внешними по отношению к программе.

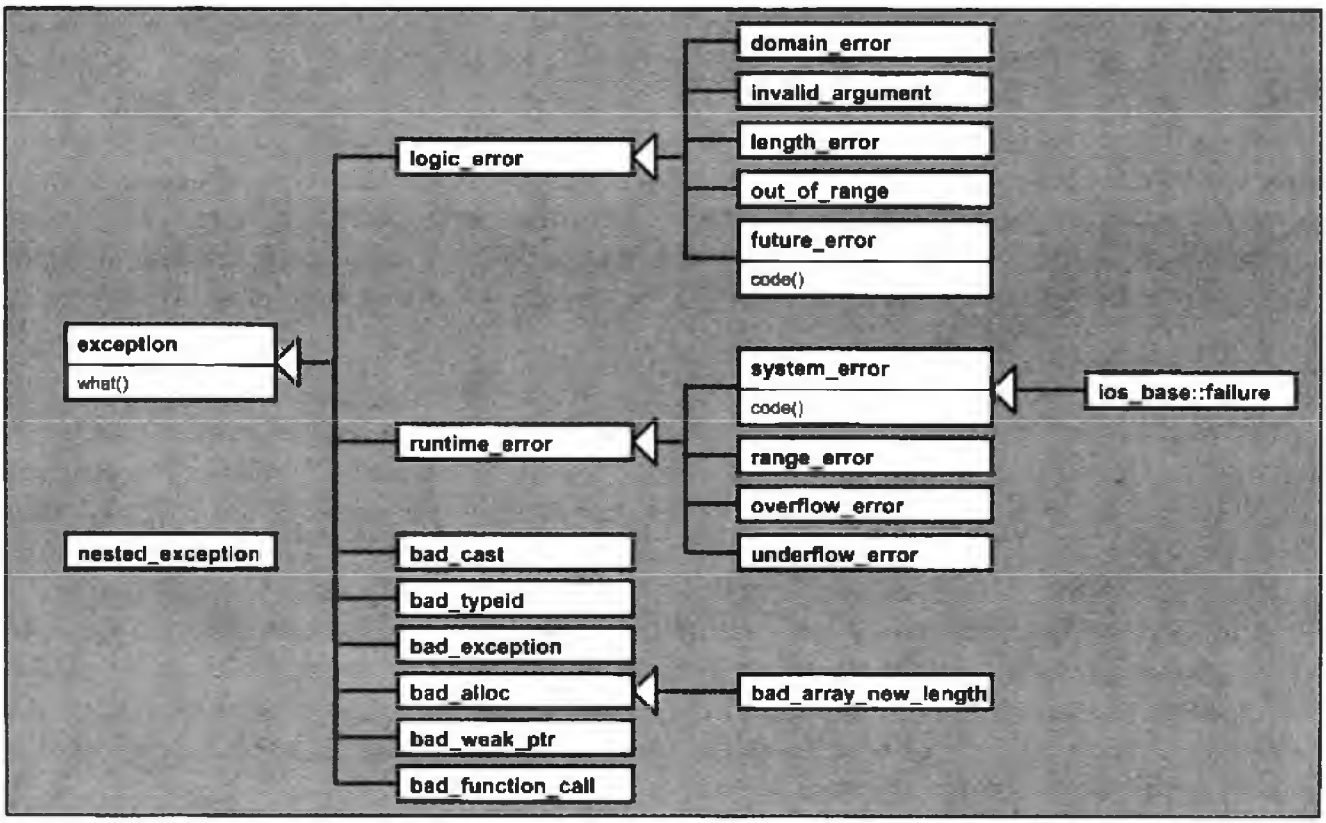


Рис. 4.1. Иерархия стандартных исключений

Классы исключений для поддержки языка

Исключения для поддержки языка используются средствами языка C++. Следовательно, было бы логичнее считать их частью базового языка, а не библиотеки. Эти исключения генерируются при неудачном выполнении некоторых операций.

- Исключение класса `bad_cast`, определенного в заголовке `<typeinfo>`, генерируется оператором `dynamic_cast`, если приведение типа ссылки во время выполнения завершается неудачей.
- Исключение класса `bad_typeid`, определенного в заголовке `<typeinfo>`, генерируется оператором `typeid`, предназначенным для идентификации типов во время выполнения. Если аргументом оператора `typeid` является нуль или нулевой указатель, генерируется исключение.
- Исключение класса `bad_exception`, определенного в заголовке `<exception>`, предназначено для обработки непредвиденных исключений. Оно может генерироваться функцией `unexpected()`, которая вызывается, если было сгенерировано исключение, не указанное в спецификации исключений соответствующей функции ((см. раздел 3.1.7).

Эти исключения также могут генерироваться библиотечными функциями. Например, исключение `bad_cast` может генерироваться функцией `use_facet()`, если аспект локализации (facet) в конкретной локализации (locale) недоступен (см. раздел 16.2.2).

Классы исключений для логических ошибок

Классы исключений для логических ошибок обычно являются производными от класса `logic_error`. Логическими называют ошибки, которые можно предотвратить, например, с помощью дополнительной проверки аргументов функции. Примерами логических ошибок являются нарушение логических предусловий или инварианта класса. Стандартная библиотека C++ содержит следующие классы для логических ошибок.

- Исключение класса `invalid_argument` используется для генерирования сообщений о некорректных аргументах, например, когда битовое множество (массив битов) инициализируется данными типа `char`, отличными от `'0'` и `'1'`.
- Исключение класса `length_error` используется для генерирования сообщений о попытке превысить установленные пределы, например, при добавлении к строке слишком большого количества символов.
- Исключение класса `out_of_range` используется для генерирования сообщений о том, что аргумент выходит за пределы допустимых значений, например, при использовании неправильного индекса в коллекциях наподобие массивов или строк.
- Исключение класса `domain_error` используется для генерирования сообщений об ошибке выхода за пределы области допустимых значений.
- В стандарте C++11 определен класс исключений `future_error` для генерирования сообщений о логических ошибках при использовании асинхронных системных вызовов (см. главу 18). Отметим, что ошибки времени выполнения в этих ситуациях приводят к генерированию исключения класса `system_error`.

В принципе, классы для логических ошибок определены в заголовке <stdexcept>, однако класс future_error определен в заголовке <future>.

Классы исключений для ошибок времени выполнения

Исключения, производные от класса runtime_error, используются для генерирования сообщений о событиях, не контролируемых программой, или об ошибках, которых трудно избежать. Стандартная библиотека C++ содержит следующие классы ошибок времени выполнения.

- Исключение класса range_error используется для генерирования сообщений об ошибках выхода за пределы допустимого диапазона во внутренних вычислениях. В соответствии со стандартом C++11 это исключение генерируется в стандартной библиотеке языка C++ при попытке выполнить преобразование широкой строки в строку байтов, и наоборот (см. раздел 16.4.4).
- Исключение класса overflow_error используется для генерирования сообщений об арифметическом переполнении. В стандартной библиотеке языка C++ это исключение может возникнуть, если битовое множество преобразовывается в целочисленное значение (см. раздел 12.5.1).
- Исключение класса underflow_error используется для генерирования сообщений о потере значимости при выполнении арифметических операций.
- В соответствии со стандартом C++11 исключение класса system_error используется для генерирования сообщений об ошибках, связанных с работой операционной системы. В стандартной библиотеке языка C++ это исключение может быть сгенерировано в контексте параллельной работы, например, классом thread, классами, управляющими "гонкой данных" и функцией async() (см. главу 18).
- Исключение класса bad_alloc, определенного в заголовке <new>, генерируется, если глобальный оператор new не достигает успеха, за исключением ситуаций, в которых используется версия оператора new, не генерирующая исключений (nothrow). Вероятно, это самое важное исключение времени выполнения, поскольку оно может появиться в любой момент в любой нетривиальной программе.
- В соответствии со стандартом C++11 исключение bad_array_new_length, производное от исключения bad_alloc, генерируется оператором new, если размер, переданный оператору new, меньше нуля или превышает установленный реализацией языка предел (т.е. это скорее логическая ошибка, а не ошибка времени выполнения).
- Исключение класса bad_weak_ptr, определенное в заголовке <memory>, генерируется при неудачном создании слабого указателя из разделяемого указателя (см. раздел 5.2.2).
- Исключение класса bad_function_call, определенного в заголовке <functional>, генерируется, когда объект function, играющий роль обертки для объекта, вызывается без указания целевого объекта (см. раздел 5.4.4).

Кроме того, часть библиотеки, реализующая ввод-вывод, генерирует исключение специального класса под названием `ios_base::failure`, определенного в заголовке `<ios>`. Исключение этого класса может быть сгенерировано, когда поток изменяет свое состояние из-за ошибки или достижения конца файла. В соответствии со стандартом C++11 этот класс является производным от класса `system_error`. Ранее он был непосредственным наследником класса `exception`. Точное поведение этого класса исключений описано в разделе 15.4.4.

Концептуально исключение `bad_alloc` можно рассматривать как системную ошибку. Однако по историческим причинам и из-за важности проблем при появлении ошибок, связанных с нехваткой памяти, проектировщики предпочитают генерировать исключение `bad_alloc`, а не `system_error`.

Вообще-то, классы ошибок времени выполнения определены в заголовочном файле `<stdexcept>`, однако класс `system_error` определен в заголовочном файле `<system_error>`.

Исключения, генерируемые стандартной библиотекой

Как уже указывалось, стандартная библиотека C++ может генерировать практически любые исключения. В частности, при выделении памяти для хранилища может генерироваться исключение `bad_alloc`.

Кроме того, поскольку компоненты библиотеки могут использовать код, написанный прикладным программистом, функции библиотеки могут опосредованно генерировать любые исключения.

Реализации стандартной библиотеки могут содержать дополнительные классы исключений, определенные на одном уровне со стандартными классами или в виде производных классов. Однако использование нестандартных классов нарушает переносимость кода, так как замена реализации стандартной библиотеки требует переработки программы. Таким образом, следует всегда использовать только стандартные классы исключений.

Заголовочные файлы для классов исключений

Классы исключений определены в разных заголовках. Для того чтобы использовать их, необходимо включить следующие заголовки:

```
#include <exception>      // для классов exception и bad_exception
#include <stdexcept>      // для большинства классов логических
                          // ошибок и ошибок времени выполнения
#include <system_error>   // для системных ошибок (C++11)
#include <new>            // для ошибок, связанных с нехваткой памяти
#include <ios>            // для ошибок ввода-вывода
#include <future>         // для ошибок, связанных с асинхронным режимом
#include <typeinfo>       // для классов bad_cast и bad_typeid
```

4.3.2. Члены классов исключений

Для обработки исключений в разделе `catch` можно использовать интерфейс, предоставляемый классами исключений. Во всех таких классах предусмотрена функция `what()`; в некоторых классах также содержится функция `code()`.

Функция-член what()

Во всех стандартных классах исключений для получения дополнительной информации, помимо сведений о самом типе исключения, можно использовать только одну функцию-член — виртуальную функцию-член what(), возвращающую строку, завершающуюся нулевым байтом.

```
namespace std {
   class exception {
      public:
        virtual const char* what() const noexcept;
        ...
   };


}
```

Содержимое строки, возвращаемой функцией what(), определяется реализацией. Отметим, что эта строка, завершающаяся нулевым байтом, может быть многобайтовой, которую можно преобразовывать и отображать как строку wstring (см. раздел 13.2.1). Строка в стиле языка C, возвращаемая функцией what(), является корректной, пока не будет уничтожен объект исключения, из которого она была получена, или пока для этого объекта не будет установлено новое значение.

Коды и условия ошибок

В классах исключений system_error и future_error имеется дополнительная функция-член, позволяющая получить подробную информацию об исключении. Но, прежде чем погружаться в детали, мы должны указать на различия между кодами и условиями ошибок.

- **Коды ошибок** — это легковесные объекты, инкапсулирующие значения кодов ошибок, которые могут зависеть от реализации. Однако некоторые коды ошибок стандартизированы.
- **Условия ошибок** — это объекты, предоставляющие переносимые абстракции для описания ошибок.

В зависимости от контекста стандартная библиотека C++ иногда определяет для исключений коды ошибок, а иногда — условия ошибок.

- Класс std::errc предоставляет *условия ошибок* для исключений класса std::system_error, соответствующие номерам стандартных системных ошибок, определенных в заголовке <cerrno> или <errno.h>.
- Класс std::io_errc предоставляет *код ошибки* для исключения класса std::ios_base::failure, генерируемого потоковыми классами в соответствии со стандартом C++11 (см. раздел 15.4.4).
- Класс std::future_errc предоставляет *коды ошибок* для исключений класса std::future_error, генерируемых библиотекой параллельного программирования (см. главу 18).

В табл. 4.1 представлены значения условий ошибок, определенные в стандартной библиотеке языка C++ для исключений класса `system_error`. Они представляют собой *перечисления с ограниченной областью видимости* (см. раздел 3.1.13), поэтому следует использовать префикс `std::errc::`. Значения этих условий необходимы для того, чтобы определить соответствующее значение `errno`, определенное в заголовке `<cerrno>` или `<errno.h>`. Это *не* код ошибки; коды ошибок обычно зависят от реализации.

Таблица 4.1. Условия ошибок для исключений класса `system_error`

Условие ошибки	Значение перечисления
address_family_not_supported	EAFNOSUPPORT
address_in_use	EADDRINUSE
address_not_available	EADDRNOTAVAIL
already_connected	EISCONN
argument_list_too_long	E2BIG
argument_out_of_domain	EDOM
bad_address	EFAULT
bad_file_descriptor	EBADF
bad_message	EBADMSG
broken_pipe	EPIPE
connection_aborted	CONNABORTED
connection_already_in_progress	EALREADY
connection_refused	ECONNREFUSED
connection_reset	ECONNRESET
cross_device_link	EXDEV
destination_address_required	EDESTADDRREQ
device_or_resource_busy	EBUSY
directory_not_empty	ENOTEMPTY
executable_format_error	ENOEXEC
file_exists	EEXIST
file_too_large	EFBIG
filename_too_long	ENAMETOOLONG
function_not_supported	ENOSYS
host_unreachable	EHOSTUNREACH
identifier_removed	EIDRM
illegal_byte_sequence	EILSEQ
inappropriate_io_control_operation	ENOTTY
interrupted	EINTR
invalid_argument	EINVAL
invalid_seek	ESPIPE
io_error	EIO
is_a_directory	EISDIR
message_size	EMSGSIZE

Продолжение табл. 4.1

Условие ошибки	Значение перечисления
network_down	ENETDOWN
network_reset	ENETRESET
network_unreachable	ENETUNREACH
no_buffer_space	ENOBUFS
no_child_process	ECHILD
no_link	ENOLINK
no_lock_available	ENOLCK
no_message_available	ENODATA
no_message	ENOMSG
no_protocol_option	ENOPROTOOPT
no_space_on_device	ENOSPC
no_stream_resources	ENOSR
no_such_device_or_address	ENXIO
no_such_device	ENODEV
no_such_file_or_directory	ENOENT
no_such_process	ESRCH
not_a_directory	ENOTDIR
not_a_socket	ENOTSOCK
not_a_stream	ENOSTR
not_connected	ENOTCONN
not_enough_memory	ENOMEM
not_supported	ENOTSUP
operation_canceled	ECANCELED
operation_in_progress	EINPROGRESS
operation_not_permitted	EPERM
operation_not_supported	EOPNOTSUPP
operation_would_block	EWOULDBLOCK
owner_dead	EOWNERDEAD
permission_denied	EACCES
protocol_error	EPROTO
protocol_not_supported	EPROTONOSUPPORT
read_only_file_system	EROFS
resource_deadlock_would_occur	EDEADLK
resource_unavailable_try_again	EAGAIN
result_out_of_range	ERANGE
state_not_recoverable	ENOTRECOVERABLE
stream_timeout	ETIME
text_file_busy	ETXTBSY
timed_out	ETIMEDOUT

Окончание табл. 4.1

Условие ошибки	Значение перечисления
too_many_files_open_in_system	ENFILE
too_many_files_open	EMFILE
too_many_links	EMLINK
too_many_symbolic_link_levels	ELOOP
value_too_large	EOVERFLOW
wrong_protocol_type	EPROTOTYPE

В табл. 4.2. приведены значения кодов ошибок, определенные стандартной библиотекой языка C++ для исключений типа `future_errc`. Они представляют собой *перечисления с ограниченной областью видимости* (см. раздел 3.1.13), поэтому следует использовать префикс `std::future_errc::`.[1]

Единственный код ошибки для исключений класса `ios_base::failure` определен значением `std::io_errc::stream`.

Таблица 4.2. Коды ошибок для исключений класса future_error

Код ошибки	Смысл
broken_promise	Разделяемое состояние аннулировано
future_already_retrieved	get_future() уже вызвана
promise_already_satisfied	Разделяемое состояние уже имеет значение/исключение или уже вызвано
no_state	Нет разделяемого состояния

Работа с кодами и условиями ошибок

Для кодов и условий ошибок в стандартной библиотеке языка C++ предусмотрены два разных класса: `std::error_code` и `std::error_condition`. Может создаться впечатление, что обработка ошибок представляет собой довольно запутанный процесс. Однако библиотека разработана так, чтобы программист всегда мог сравнивать коды ошибок с условиями ошибок, используя как объекты, так и значения перечислений.

Например, для любого объекта ошибки `ec` типа `std::error_code` или `std::error_condition` можно выполнить следующие операторы:

```
if (ec == std::errc::invalid_argument) { // проверка конкретного
                                         // условия ошибки
   ...
}
if (ec == std::future_errc::no_state) {  // проверка конкретного
                                         // кода ошибки
```

[1] Отметим, что в стандарте C++11 коды будущих ошибок явно определены значением `future_errc::broken_promise` и равны нулю. Но, поскольку нулевой код ошибки обычно означает отсутствие ошибок, это следует считать неправильным решением. Для того чтобы исправить эту проблему, все значения кодов для будущих ошибок определены как зависящие от реализации.

```
   ...
}
```

Таким образом, при обработке ошибок с помощью конкретных кодов или условий ошибок разница между ними несущественна.

Для работы с кодами и условиями ошибок класс `std::system_error`, а также его производный класс `std::ios_base::failure` и класс `std::_future_error` содержат дополнительную невиртуальную функцию-член `code()`, возвращающую объект класса `std::error_code`.[2]

```
namespace std {
   class system_error : public runtime_error {
      public:
         virtual const char* what() const noexcept;
         const error_code& code() const noexcept;
         ...
};

   class future_error : public logic_error {
      public:
         virtual const char* what() const noexcept;
         const error_code& code() const noexcept;
         ...
   };
}
```

В классе `error_code` предусмотрены функции-члены для получения детальной информации об ошибке.

```
namespace std {
   class error_code {
      public:


         const error_category& category() const noexcept;
         int value() const noexcept;
         string message() const;
         explicit operator bool() const noexcept;
         error_condition default_error_condition() const noexcept;
         ...
   };
}
```

Этот интерфейс основан на нескольких соображениях.

- Разные библиотеки могут использовать одни и те же целочисленные значения для разных кодов ошибок. По этой причине каждая ошибка имеет категорию и значение. Каждое значение является уникальным и имеет конкретный смысл только внутри категории.

[2] Строго говоря, эти объявления находятся в разных заголовочных файлах, а функция `what()`, хотя и не объявлена как виртуальная, наследует виртуальность из базового класса.

- Функция-член `message()` возвращает соответствующее сообщение, которое обычно является частью результата работы функции-члена `what()` для всех исключений, хотя это и не требуется.
- Операторная функция-член `operator bool()` сообщает, установлен ли код ошибки (значение 0 означает, что ошибки нет). При перехвате исключения этот оператор обычно возвращает значение `true`.
- Функция-член `default_error_condition()` возвращает соответствующий объект `error_condition`, также содержащий функции-члены `category()`, `value()`, `message()` и `operator bool()`.

```
namespace std {
   class error_condition {
      public:
         const error_category& category() const noexcept;
         int value() const noexcept;
         string message() const;
         explicit operator bool() const noexcept;
         ...
      };
   }
```

Класс `std::error_category` предоставляет следующий интерфейс:

```
namespace std {
   class error_category {
      public:
      virtual const char* name() const noexcept = 0;
      virtual string message (int ev) const = 0;
      virtual error_condition default_error_condition (int ev)
                                                    const noexcept;
      bool operator == (const error_category& rhs) const noexcept;
      bool operator != (const error_category& rhs) const noexcept;
      ...
   };
}
```

где функция `name()` возвращает имя категории, а функции `message()` и `default_error_condition()` возвращают сообщение и условие ошибки, заданное по умолчанию в соответствии с переданным значением (т.е. то, что вызывают соответствующие функции-члены класса `error_code`). Операторы == и != позволяют сравнивать категории ошибок.

В стандартной библиотеке шаблонов языка C++ определены следующие имена категорий:

- `"iostream"` — для исключений потоков ввода-вывода типа `ios_base::failure`;
- `"generic"` — для системных исключений типа `system_error`, если их значения соответствуют значениям ошибок в стандарте POSIX;
- `"system"` — для системных исключений типа `system_error`, если их значения не соответствуют значениям ошибок в стандарте POSIX;
- `"future"` — для исключений типа `future_error`.

Для каждой категории предусмотрены глобальные функции, возвращающие категорию[3].

```
const error_category& generic_category() noexcept; // в <system_errror>
const error_category& system_category() noexcept;  // в <system_error>
const error_category& iostream_category();         // в <ios>
const error_category& future_category() noexcept;  // в <future>
```

Таким образом, из объекта кода ошибки e можно вызвать функцию-член, чтобы определить, является ли он кодом сбоя ввода-вывода.

```
if (e.code().category() == std::iostream_category())
```

Следующая программа демонстрирует, как с помощью обобщенной функции можно обрабатывать (в данном случае выводить на экран) разные исключения:

```
// util/exception.hpp
#include <exception>
#include <system_error>
#include <future>
#include <iostream>

template <typename T>
void processCodeException (const T& e)
{
   using namespace std;
   auto c = e.code();
   cerr << "- category: " << c.category().name() << endl;
   cerr << "- value: " << c.value() << endl;
   cerr << "- msg: " << c.message() << endl;
   cerr << "- def category: "
        << c.default_error_condition().category().name() << endl;
   cerr << "- def value: "
        << c.default_error_condition().value() << endl;
   cerr << "- def msg: "
        << c.default_error_condition().message() << endl;
}

void processException()
{
   using namespace std;
   try {
       throw; // повторно генерируем исключение,
              // чтобы обработать его здесь
   }
   catch (const ios_base::failure& e) {
       cerr << "I/O EXCEPTION: " << e.what() << endl;
       processCodeException(e);
   }
   catch (const system_error& e) {
      cerr << "SYSTEM EXCEPTION: " << e.what() << endl;
```

[3] Вероятно, по недосмотру функция-член `iostream_category()` объявлена без ключевого слова `noexcept`.

```
      processCodeException(e);
   }
   catch (const future_error& e) {
      cerr << "FUTURE EXCEPTION: " << e.what() << endl;
      processCodeException(e);
   }
   catch (const bad_alloc& e) {
      cerr << "BAD ALLOC EXCEPTION: " << e.what() << endl;
   }
   catch (const exception& e) {
      cerr << "EXCEPTION: " << e.what() << endl;
   }
   catch (...) {
      cerr << "EXCEPTION (unknown)" << endl;
   }
}
```

Это позволяет обработать исключения следующим образом:

```
try {
   ...
}
catch (...) {
   processException();
}
```

Другие члены

Остальные члены классов стандартных исключений создают, копируют, присваивают и уничтожают объекты исключений.

Отметим, что кроме функций-членов `what()` и `code()` ни в одном классе стандартных исключений нет дополнительного члена, описывающего вид исключения. Например, нет переносимого способа выяснить контекст исключения или неправильный индекс при ошибке выхода за пределы диапазона. Таким образом, единственным переносимым способом определить вид исключения является вывод сообщения, возвращаемого функцией `what()`.

```
try {
   ...
}
catch (const std::exception& error) {
   // выводим на печать сообщение об ошибке, определенное реализацией
   std::cerr << error.what() << std::endl;
   ...
}
```

Единственной возможной альтернативой может быть интерпретация точного типа исключения. Например, если было сгенерировано исключение `bad_alloc`, программа может попытаться получить больше памяти.

4.3.3. Передача исключений с помощью класса exception_ptr

В соответствии со стандартом C++11 стандартная библиотека языка C++ обеспечивает возможность хранить исключения в объектах типа `exception_ptr`, чтобы обработать их позднее или в другом контексте.

```
#include <exception>

std::exception_ptr eptr; // объект для хранения исключений (или nullptr)

void foo ()
{
   try {
       throw ...;
   }
   catch (...) {
       eptr = std::current_exception(); // сохраняем исключения
                                        // для дальнейшей обработки
   }
}

void bar ()
{
   if (eptr != nullptr) {
       std::rethrow_exception(eptr);    // обрабатываем сохраненное
                                        // исключение
   }
}
```

Функция-член `current_exception()` возвращает объект класса `exception_ptr`, ссылающийся на обрабатываемое исключение. Значение, возвращаемое функцией-членом `current_exception()`, является корректным, пока на него ссылается объект класса `exception_ptr`. Функция-член `rethrow_exception()` повторно генерирует сохраненное исключение, чтобы функция-член `bar()` работала так, будто в ней функция-член `foo()` сгенерировала первичное исключение.

Эта функциональная возможность особенно полезна при передаче исключения между потоками (см. раздел 18.2.1).

4.3.4. Генерирование стандартных исключений

Программист может генерировать стандартные исключения в своих библиотеках и программах. Все классы стандартных ошибок, включая логические ошибки и ошибки времени выполнения, содержащие интерфейсную функцию-член `what()`, имеют конструктор только с аргументом `std::string` и (начиная со стандарта C++11) `const char*`. Значение, переданное в объект этого класса, превращается в описание, возвращаемое функцией `what()`. Например, класс `logic_error` определен следующим образом:

```
namespace std {
   class logic_error : public exception {
     public:
```

```
        explicit logic_error (const string& whatString);
        explicit logic_error (const char* whatString); // since C++11
        ...
    };
}
```

Класс `std::system_error` предусматривает возможность создания объекта исключения с помощью передачи кода ошибки, строки для функции `what()` и необязательной категории.

```
namespace std {
    class system_error : public runtime_error {
      public:
        system_error (error_code ec, const string& what_arg);
        system_error (error_code ec, const char* what_arg);
        system_error (error_code ec);
        system_error (int ev, const error_category& ecat,
                      const string& what_arg);
        system_error (int ev, const error_category& ecat,
                      const char* what_arg);
        ...
    };
}
```

Для создания объекта `error_code` предусмотрены вспомогательные функции-члены `make_error_code()`, получающие только значение кода ошибки.

Класс `std::ios_base::failure` имеет конструктор, получающий строку, которая является аргументом функции `what()`, и (начиная со стандарта C++11) необязательный объект `error_code`. Класс `std::future_error` имеет лишь конструктор, получающий один объект класса `error_code`.

Таким образом, генерирование стандартного исключения не представляет затруднений.

```
throw std::out_of_range ("out_of_range (somewhere, somehow)");

throw
    std::system_error (std::make_error_code(std::errc::invalid_argument),
                       "argument ... is not valid");
```

Отметим, что нельзя генерировать исключения базового класса `exception` и любые исключения, предназначенные для языковой поддержки (`bad_cast`, `bad_typeid`, `bad_exception`).

4.3.5. Наследование классов стандартных исключений

Другая возможность использования классов стандартных исключений в своем коде заключается в определении специального класса исключений, производного прямо или косвенно от класса `exception`. Для этого следует убедиться в том, что механизмы функции `what()` или `code()` действительно работают, что вполне возможно, потому что функция-член `what()` является виртуальной. В качестве примера см. класс `Stack` в разделе 12.1.3.

4.4. Вызываемые объекты

В разных местах стандартной библиотеки языка C++ используется термин *вызываемый объект* (callable object), обозначающий объекты, которые так или иначе можно использовать для вызова некоей функциональности, например:

- функция, получающая дополнительные аргументы в виде аргументов функции;
- указатель на функцию-член, которая вызывается для объекта, передаваемого как первый дополнительный аргумент (должен быть ссылкой или указателем), и получает остальные аргументы как параметры функции-члена;
- функциональный объект (оператор () передаваемого объекта), которому передаются дополнительные аргументы;
- лямбда-функция (см. раздел 3.1.10), которая, строго говоря, является разновидностью функционального объекта.

Например:

```
void func (int x, int y);

auto l = [] (int x, int y) {
     ...
};

class C {
   public:
      void operator () (int x, int y) const;
      void memfunc (int x, int y) const;
   };

int main()
{
   C c;
   std::shared_ptr<C> sp(new C);

   // Функция bind() использует вызываемые
   // объекты для связывания аргументов:
   std::bind(func,77,33)();           // вызывает: func(77,33)
   std::bind(l,77,33)();              // вызывает: l(77,33)
   std::bind(C(),77,33)();            // вызывает: C::operator()(77,33)
   std::bind(&C::memfunc,c,77,33)();  // вызывает: c.memfunc(77,33)
   std::bind(&C::memfunc,sp,77,33)(); // вызывает: sp->memfunc(77,33)

   // Функция async() использует вызываемые объекты
   // для запуска задач (в фоновом режиме):
   std::async(func,42,77);            // вызывает: func(42,77)
   std::async(l,42,77);               // вызывает: l(42,77)
   std::async(c,42,77);               // вызывает: c.operator()(42,77)
   std::async(&C::memfunc,&c,42,77);  // вызывает: c.memfunc(42,77)
   std::async(&C::memfunc,sp,42,77);  // вызывает: sp->memfunc(42,77)
}
```

Для передачи объекта, для которого вызывается функция-член, можно использовать даже интеллектуальные указатели (см. раздел 5.2). Функция `std::bind()` описана в разделе 10.2.2, а функция `std::async()` — в разделе 18.1.

Для объявления *вызываемых объектов* в общем случае можно использовать класс `std::function<>` (см. раздел 5.4.4).

4.5. Параллельное программирование и многопоточность

До появления стандарта C++11 в языке C++ и его стандартной библиотеке не было поддержки параллельного программирования, хотя реализации могли предоставлять определенные гарантии. С появлением стандарта C++11 ситуация изменилась. В ядро языка и в библиотеку включены усовершенствования для поддержки параллельного программирования.

Например, для параллельной работы в ядро языка внесено несколько новшеств.

- Теперь существует модель памяти, гарантирующая, что обновления двух разных объектов двумя разными потоками происходят независимо друг от друга. До стандарта C++11 не было гарантии, что запись переменной типа `char` в одном потоке не повлияет на запись *другой* переменной типа `char` в другом потоке (см. раздел *"The memory model"* в работе [Stroustrup:C++0x]).
- Появилось новое ключевое слово `thread_local`, необходимое для определения переменных и объектов, зависящих от потока.

В библиотеку внесены следующие изменения:

- определенные гарантии безопасности потоков;
- вспомогательные классы и функции для параллельного программирования (запуск и синхронизация нескольких потоков).

Эти вспомогательные классы и функции рассматриваются в главе 18. Хотя гарантии, предоставляемые этими классами и функциями, обсуждаются на всем протяжении книги, здесь целесообразно сделать их краткий обзор.

Общие гарантии параллельного программирования в стандартной библиотеке C++

Рассмотрим общие ограничения, касающиеся поддержки параллельного программирования и многопоточности в стандартной библиотеке C++ в соответствии со стандартом C++11.

- В общем случае совместное использование библиотечного объекта несколькими потоками — при условии, что хотя бы один поток модифицирует этот объект, — может привести к непредсказуемым последствиям. Процитируем стандарт: "*Модификация объекта стандартного библиотечного типа, совместно используемого потоками, порождает риск возникновения непредсказуемых последствий, если*

объекты этого типа не определены явно как разделяемые без гонки данных, или если пользователь не предусмотрел блокировочный механизм".

- Особенно рискованная ситуация возникает, когда объект создается в одном потоке, а используется в другом. Аналогично непредсказуемые последствия могут возникнуть при уничтожении объекта в одном потоке, в то время как он продолжает использоваться в другом. Отметим, что это относится даже к объектам, предназначенным для синхронизации потоков.

Наиболее важные места, в которых *поддерживается* параллельный доступ к библиотечным объектам, описаны ниже.

- Для контейнеров STL (см. главу 7) и адаптеров контейнеров (см. главу 12) предусмотрены следующие гарантии.
  - Разрешается параллельный доступ только для чтения. Это явно подразумевает вызов неконстантных функций-членов `begin()`, `end()`, `rbegin()`, `rend()`, `front()`, `back()`, `data()`, `find()`, `lower_bound()`, `upper_bound()`, `equal_range()`, `at()` и, за исключением ассоциативных контейнеров, оператора `[]` и доступа с помощью итераторов, если они не модифицируют контейнеры.
  - Разрешается параллельный доступ к *разным элементам* одного и того же контейнера (за исключением класса `vector<bool>`). Таким образом, разные потоки могут параллельно читать и/или записывать разные элементы одного и того же контейнера. Например, каждый поток может что-нибудь обрабатывать и сохранять результат в "своем" элементе совместно используемого вектора.
- При форматированном вводе и выводе в стандартный поток, синхронизированном с механизмом ввода-вывода из языка C (см. раздел 15.14.1), возможен параллельный доступ, хотя он и может привести к чередующимся символам. По умолчанию это относится к объектам `std::cin`, `std::cout` и `std::cerr`. Однако для строковых и файловых потоков, а также буферов потоков параллельный доступ приводит к непредсказуемым результатам.
- Параллельные вызовы функций `atexit()` и `at_quick_exit()` (см. раздел 5.8.2) синхронизированы. Это относится и к функциям `set_new_handler()`, `set_unexpected()`, `set_terminate()` и их `get_`-аналогам. Функция `getenv()` также синхронизирована.
- Для всех функций-членов распределителя памяти, заданного по умолчанию (см. главу 19), за исключением деструкторов, параллельный доступ является синхронизированным.

Кроме того, стандартная библиотека языка C++ гарантирует, что в ней нет скрытых побочных эффектов, нарушающих параллельный доступ к разным объектам. Таким образом, стандартная библиотека языка C++

- не обращается к досягаемым объектам, которые не требуются для конкретной операции;
- не может скрытно вводить совместно используемые статические объекты без синхронизации;

- позволяет реализациям распараллеливать операции только при условии, что при этом не возникают заметные побочные эффекты. Тем не менее не следует забывать о фактах, изложенных в разделе 18.4.2.

4.6. Распределители памяти

В некоторых частях стандартной библиотеки языка C++ используются специальные объекты для выделения и освобождения памяти, которые называются *распределителями памяти* (allocators). Они представляют собой определенную модель памяти и используются как абстракции, преобразующие запросы на выделение памяти в физическую операцию ее выделения. Одновременное использование разных объектов распределителя памяти позволяет использовать в своей программе разные модели памяти.

Изначально распределители памяти появились в библиотеке STL для решения тяжелой проблемы, связанной с разными типами указателей для разных моделей памяти (например, `near`, `far` и `huge`) на машинах PC. В настоящее время на основе распределителей памяти разработаны технические решения, не требующие изменения интерфейса и использующие разные модели памяти, такие как совместно используемая память, сбор мусора, объектно-ориентированные базы данных. Однако эти решения возникли относительно недавно и еще не получили широкого распространения (вероятно, со временем ситуация изменится).

В стандартной библиотеке C++ определен *распределитель памяти по умолчанию*.

```
namespace std {
   template <class T>
   class allocator;
}
```

Распределитель памяти по умолчанию используется во всех тех ситуациях, когда распределитель может передаваться в качестве аргумента. Он вызывает стандартные механизмы выделения и освобождения памяти, т.е. операторы `new` и `delete`. Однако, когда и как эти операторы должны вызываться, остается неопределенным. Таким образом, конкретная реализация распределителя памяти по умолчанию может, например, выполнять внутреннее кеширование выделяемой памяти.

В большинстве программ используется распределитель памяти по умолчанию, но некоторые библиотеки предоставляют специальные распределители памяти, которые просто передаются в виде аргументов. Необходимость самостоятельно программировать распределители памяти возникает очень редко. На практике обычно вполне достаточно использовать распределитель памяти по умолчанию. Мы отложим обсуждение распределителей памяти до главы 19, в которой рассматриваются не только распределители памяти, но и их интерфейсы.

Глава 5

Вспомогательные средства

В этой главе описываются общие вспомогательные средства стандартной библиотеки языка C++. Они представляют собой небольшие и простые классы или функции, выполняющие часто возникающие задачи.

- Классы `pair<>` и `tuple<>`.
- Классы интеллектуальных указателей (`shared_ptr<>` и `unique_ptr`).
- Числовые пределы[1].
- Свойства типов и утилиты для работы с типами.
- Вспомогательные функции (например `min()`, `max()` и `swap()`).
- Класс `ratio<>`[1].
- Часы и таймеры.
- Некоторые важные функции языка C.

Большинство этих утилит, но не все, описаны в стандарте языка C++ (Chapter 20, "General Utilities). Остальные описаны вместе с главными компонентами библиотеки либо потому, что они используются в основном конкретным компонентом, либо по историческим причинам. Например, некоторые общие вспомогательные функции определены в заголовке `<algorithm>`, хотя они не являются алгоритмами в смысле библиотеки STL (которые рассматриваются в главе 6).

Некоторые из этих утилит используются также в стандартной библиотеке языка C++. Например, тип `pair<>` используется всюду, где два значения необходимо интерпретировать как одно целое, — например, если функция должна возвращать два значения или если элементами контейнера являются пары "ключ–значение",— а свойства типов используются при выполнении сложных преобразований типов.

5.1. Пары и кортежи

В библиотеку C++98, первую версию стандартной библиотеки языка C++, был включен простой класс, предназначенный для работы с парами значений разного типа, не требующими определения специального класса. В библиотеке C++98 этот класс использовался тогда, когда стандартные функции возвращали пару значений или когда элементами контейнера были пары "ключ/значение".

---

[1] На это можно возразить, что числовые пределы и класс `ratio<>` следует рассматривать в главе 17, посвященной числам, но эти классы используются и в других частях библиотеки, поэтому я решил описать их здесь.

В документе TR1 был введен в рассмотрение класс кортежей, расширяющий эту концепцию до набора произвольного, но конечного количества элементов. Конкретные реализации позволяют кортежам содержать до десяти элементов разного типа на разных платформах.

В стандарте C++11 класс кортежей был реализован заново на основе концепции вариативных шаблонов (см. раздел 3.1.9). Теперь существует стандартный тип кортежей для неоднородных коллекций произвольного размера. Кроме того, для работы с парами элементов по-прежнему предназначен класс `pair`, который может использоваться в сочетании с двухэлементными кортежами.

В то же время класс `pair` в стандарте C++11 был значительно расширен, что в определенном смысле соответствует расширениям возможностей самого языка C++ и его библиотеки.

5.1.1. Пары

Класс `pair` интерпретирует два значения как одно целое. Этот класс используется в нескольких местах стандартной библиотеки языка C++. В частности, контейнерные классы `map`, `multimap`, `unordered_map` и `unordered_multimap` используют класс `pair` для управления своими элементами, представляющими собой пары "ключ/значение" (см. раздел 7.8). Другими примерами использования класса `pair` являются функции, возвращающие два значения, например `minmax()` (см. раздел 5.5.1).

Структура `pair` определена в заголовке `<utility>` и предусматривает операции, приведенные в табл. 5.1. В принципе объекты класса `pair<>` можно создавать, копировать/присваивать/обменивать, а также сравнивать. Кроме того, существуют определения типов `first_type` и `second_type` для типов первого и второго значений.

Таблица 5.1. Операции над парами

Операция	Действие
`pair<T1,T2>` *p*	Конструктор по умолчанию; создает пару значений с типами `T1` и `T2`, инициализированных своими конструкторами, заданными по умолчанию
`pair<T1,T2>` *p*(*val1*, *val2*)	Создает пару значений с типами `T1` и `T2`, инициализированных значениями *val1* и *val2*
`pair<T1,T2>` *p*(*rv1*, *rv2*)	Создает пару значений с типами `T1` и `T2`, инициализированных rvalue-ссылками *rv1* и *rv2*[2]
`pair<T1,T2>` *p*(`piecewise_construct`, *t1*, *t2*)	Создает пару значений с типами `T1` и `T2`, инициализированных элементами кортежей *t1* и *t2*
`pair<T1,T2>` *p*(*p2*)	Копирующий конструктор; создает объект `p` как копию объекта *p2*
`pair<T1,T2>` *p*(*rv*)	Перемещающий конструктор; перемещает содержимое объекта `rv` в объект `p` (возможны неявные преобразования типов)

[2] Здесь и далее буквами *rv* обозначены не значения, а rvalue-ссылки, описываемые в разделе 3.1.5. — *Примеч. консульт.*

Окончание табл. 5.1

Операция	Действие
p = *p2*	Присваивает значения объекта p2 объекту p (в соответствии со стандартом C++11 возможны неявные преобразования типов)
p = *rv*	Перемещающее присваивание значений из объекта rv объекту p (введено в стандарте C++11; возможны неявные преобразования типов)
p.first	Возвращает первое значение пары (прямой доступ к члену)
p.second	Возвращает второе значение пары (прямой доступ к члену)
get<0>(*p*)	Эквивалент операции *p*.first (начиная со стандарта C++11)
get<1>(*p*)	Эквивалент операции *p*.second (начиная со стандарта C++11)
p1 == *p2*	Возвращает результат сравнения на равенство объектов *p1* и *p2* (эквивалент операции *p1*.first==*p2*.first && *p1*.second==*p2*.second)
p1 != *p2*	Возвращает результат сравнения на неравенство объектов *p1* и *p2* (!(*p1*==*p2*))
p1 < *p2*	Возвращает результат проверки, является ли объект *p1* меньшим объекта *p2* (сравнивает элементы first, а если они равны — элементы second обоих объектов)
p1 > *p2*	Возвращает результат проверки, является ли объект *p1* большим объекта *p2* (*p2*<*p1*)
p1 <= *p2*	Возвращает результат проверки, не превышает ли объект *p1* объект *p2* (!(*p2*<*p1*))
p1 >= *p2*	Возвращает результат проверки, не превышает ли объект *p2* объект *p1* (!(*p1*<*p2*))
p1.swap(*p2*)	Меняет местами данные в объектах *p1* и *p2* (по стандарту C++11)
swap(*p1*,*p2*)	То же самое (как глобальная функция) (по стандарту C+11)
make_pair(*val1*,*val2*)	Возвращает пару с типами и значениями аргументов *val1* и *val2*

Доступ к элементам

Для обработки значений, являющихся элементами класса pair, в нем предусмотрены специальные члены, обеспечивающие прямой доступ к ним. Фактически тип pair объявлен с помощью ключевого слова struct, а не class, поэтому его члены являются открытыми.

```
namespace std {
   template <typename T1, typename T2>
   struct pair {
      // член
      T1 first;
      T2 second;
      ...
```

```
    };
}
```

Например, чтобы реализовать обобщенную шаблонную функцию, записывающую в поток пару значений, необходимо написать следующую программу:[3]

```
// обобщенный оператор вывода пар (ограниченное решение)
template <typename T1, typename T2>
std::ostream& operator << (std::ostream& strm,
                           const std::pair<T1,T2>& p)
{
   return strm << "[" << p.first << "," << p.second << "]";
}
```

Кроме того, в стандарте C++11 предусмотрен интерфейс, ориентированный на кортежи (см. раздел 5.1.2). Таким образом, можно использовать член `tuple_size<>::value` для выяснения количества элементов, `tuple_element<>::type` — для распознавания типа конкретного элемента и функцию `get()` — для доступа к элементам `first` или `second`:

```
typedef std::pair<int,float> IntFloatPair;
IntFloatPair p(42,3.14);

std::get<0>(p)                              // возвращает p.first
std::get<1>(p)                              // возвращает p.second
std::tuple_size<IntFloatPair>::value        // возвращает 2
std::tuple_element<0,IntFloatPair>::type    // возвращает int
```

Конструкторы и операторы присваивания

Конструктор по умолчанию создает пару значений, инициализированных конструкторами по умолчанию их типов. По правилам языка явный вызов конструктора по умолчанию также инициализирует данные элементарных типов, таких как `int`. Таким образом, объявление

```
std::pair<int,float> p; // инициализирует p.first и p.second нулем
```

инициализирует значения объекта `p` с помощью конструкторов `int()` и `float()`, которые в обоих случаях возвращают нуль. Описание правил явной инициализации для элементарных типов описано в разделе 3.2.1.

Копирующий конструктор предусмотрен в двух версиях — для копирования пары того же типа, что и создаваемая, а также как шаблонный член, используемый при необходимости неявного преобразования типа. Если типы элементов совпадают, вызывается обычный неявно генерируемый копирующий конструктор[4]. Рассмотрим пример:

```
void f(std::pair<int,const char*>);
void g(std::pair<const int,std::string>);
```

---

[3] Отметим, что этот оператор вывода не работает без механизма *ADL* (*argument-dependent lookup — поиск с учетом аргументов*) (см. раздел 15.11.1).

[4] Шаблонный конструктор не перекрывает неявно генерируемый копирующий конструктор. Подробности см. в разделе 3.2.

```
...
void foo() {
   std::pair<int,const char*> p(42,"hello");
   f(p); // OK: вызывает неявно генерируемый копирующий конструктор
   g(p); // OK: вызывает шаблонный конструктор
}
```

В соответствии со стандартом C++11 класс `pair<>`, использующий тип, имеющий только неконстантный копирующий конструктор, больше не компилируется[5]:

```
class A
{
   public:
      ...
      A(A&); // копирующий конструктор с неконстантной ссылкой
      ...
};

std::pair<A,int> p; // Ошибка по стандарту C++11
```

В соответствии со стандартом C++11 оператор присваивания также предусмотрен как шаблонный член класса `pair`, благодаря чему появляется возможность выполнять неявные преобразования. Кроме того, поддерживается семантика перемещения — перемещение первого и второго элементов.

Создание по частям

В классе `pair<>` предусмотрены три конструктора для инициализации членов `first` и `second` их начальными значениями.

```
namespace std {
   template <typename T1, typename T2>
   struct pair {
      ...
      pair(const T1& x, const T2& y);
      template<typename U, typename V> pair(U&& x, V&& y);
      template <typename... Args1, typename... Args2>
         pair(piecewise_construct_t,
              tuple<Args1...> first_args,
              tuple<Args2...> second_args);
      ...
   };
}
```

Первые два конструктора работают как обычно: передают один аргумент для члена `first` и другой для члена `second`, в том числе с поддержкой семантики перемещения и неявных преобразований типа. Однако третий конструктор имеет особый характер. Он позволяет передавать два кортежа — объекты, содержащие переменное количество элементов разных типов (см. раздел 5.1.2), — но обрабатывает их иначе. Обычно при передаче одного или двух кортежей первые два конструктора позволяют инициализировать

[5] Благодарю Даниэля Крюглера (Daniel Krügler) за это разъяснение.

пару, в которой члены `first` и/или `second` являются кортежами. В то же время третий конструктор использует кортежи для передачи их *элементов* конструкторам членов `first` и `second`. Для этого необходимо передать дополнительный параметр `std::piecewise_construct` в качестве первого аргумента. Рассмотрим пример:

```
// util/pair1.cpp

#include <iostream>
#include <utility>
#include <tuple>
using namespace std;

class Foo {
   public:
      Foo (tuple<int, float>) {
         cout << "Foo::Foo(tuple)" << endl;
      }
      template <typename... Args>
      Foo (Args... args) {
          cout << "Foo::Foo(args...)" << endl;
      }
};

int main()
{
   // создаем кортеж t:
   tuple<int,float> t(1,2.22);

   // передаем конструктору Foo кортеж как единое целое:
   pair<int,Foo> p1 (42, t);

   // передаем конструктору Foo элементы кортежа:
   pair<int,Foo> p2 (piecewise_construct, make_tuple(42), t);
}
```

Программа выводит на экран следующие результаты:

```
Foo::Foo(tuple)
Foo::Foo(args...)
```

Только если параметр `std::piecewise_construct` передается как первый аргумент, класс `Foo` вынужденно использует конструктор, получающий элементы кортежа (`int` и `float`), а не кортеж как единое целое. В этом примере это означает, что будет вызван конструктор класса `Foo` с переменным количеством аргументов. Если в классе предусмотрен конструктор `Foo::Foo(int,float)`, то будет вызван именно он.

Легко видеть, что для такого поведения необходимо, чтобы оба аргумента были кортежами, поэтому первый аргумент, `42`, был явно преобразован в кортеж с помощью функции `make_tuple()` (вместо этого можно было бы передать объект `std::tuple(42)`).

Обратите внимание на то, что именно эта форма инициализации требуется, чтобы новый элемент был вставлен в (неупорядоченное) отображение или мультиотображение с помощью функции `emplace()` (см. разделы 7.8.2 и 7.9.3).

Вспомогательная функция make_pair()

Шаблонная функция make_pair() позволяет создавать пары значений, не указывая типы элементов явно[6]. Например, вместо кода

```
std::pair<int,char>(42,'@')
```

можно написать

```
std::make_pair(42,'@')
```

До появления стандарта C++11 функция просто объявлялась и определялась следующим образом:

```
namespace std {
   // создает пару значений, просто возвращая эти значения
   template <template T1, template T2>
   pair<T1,T2> make_pair (const T1& x, const T2& y) {
      return pair<T1,T2>(x,y);
   }
}
```

Однако в стандарте C++11ситуация усложнилась, поскольку этот класс теперь может реализовывать полезную семантику перемещения. Итак, теперь стандартная библиотека языка C++ содержит следующее определение функции make_pair():

```
namespace std {
   // создает пару значений, просто возвращая эти значения
   template <template T1, template T2>
   pair<V1,V2> make_pair (T1&& x, T2&& y);
}
```

где детали возвращаемых значений и их типов V1 и V2 зависят от типов аргументов x и y.

Не вдаваясь в детали, стандарт определяет, что функция make_pair() по возможности использует семантику перемещения, а в противном случае применяет семантику копирования. Кроме того, он *ослабляет* аргументы так, чтобы, например, вызов make_pair("a","xy") создавал объект pair<const char*,const char*>, а не pair<const char[2],const char[3]> (см. раздел 5.4.2).

Функция make_pair() позволяет удобно передавать два значения пары непосредственно функции, принимающей аргумент типа pair. Рассмотрим такой пример:

```
void f(std::pair<int,const char*>);
void g(std::pair<const int,std::string>);
...
void foo() {
   f(std::make_pair(42,"empty")); // передаем два значения как пару
   g(std::make_pair(42,"chair")); // передаем два значения как пару
                                  // с преобразованиями типов
}
```

[6] Использование функции make_pair() не требует дополнительных затрат времени исполнения. Компилятор должен всегда оптимизировать любые дополнительные затраты.

Как показано в данном примере, функция `make_pair()` работает, даже если типы аргументов не совпадают точно, поскольку шаблонный конструктор обеспечивает неявное преобразование типа. Эта возможность часто требуется при работе с отображениями или мультиотображениями (см. раздел 7.8.2).

Обратите внимание, что, в соответствии со стандартом C++11, в качестве альтернативы можно использовать списки инициализации:

```
f({42,"empty"}); // передаем два значения как пару
g({42,"chair"}); // передаем два значения как пару с преобразованиями типов
```

Однако выражение, содержащее явное преобразование типа, имеет преимущество, потому что результирующий тип пары не выводится из типа значений. Например, выражение

```
std::pair<int,float>(42,7.77)
```

не эквивалентно выражению

```
std::make_pair(42,7.77)
```

Последнее выражение создает пару, в которой второе значение имеет тип `double` (не полностью определенные литералы с плавающей точкой имеют тип `double`). Точный тип может оказаться важным при использовании перегруженных функций или шаблонов. Эти функции или шаблоны, например, могут предоставлять версии как для типа `float`, так и для типа `double`, чтобы повысить эффективность программы.

В соответствии с новой семантикой стандарта C++11 существует возможность влиять на тип результата, возвращаемого функцией `make_pair()`, заставляя ее применять семантику перемещения или ссылок. Для семантики перемещения следует просто вызвать функцию `std::move()`, объявляя, что передаваемый аргумент больше не используется.

```
std::string s, t;
...
auto p = std::make_pair(std::move(s),std::move(t));
... // объекты s и t больше не используются
```

Для того чтобы применить семантику ссылок, следует вызывать функцию `ref()`, использующую ссылочный тип, или функцию `cref()`, использующую константный ссылочный тип (они определены в заголовке `<functional>`; см. раздел 5.4.3). Например, в следующих операторах пара дважды ссылается на объект типа `int`, так что в итоге переменная `i` имеет значение 2:

```
#include <utility>
#include <functional>
#include <iostream>

int i = 0;
auto p = std::make_pair(std::ref(i),std::ref(i)); // создает pair<int&,int&>
++p.first;                                        // увеличивает i
++p.second;                                       // снова увеличивает i
std::cout << "i: " << i << std::endl;             // выводит i: 2
```

В соответствии со стандартом C++11, для того чтобы извлекать значения из пары, можно также использовать интерфейс `tie()`, определенный в заголовке `<tuple>`.

```
#include <utility>
#include <tuple>
#include <iostream>

std::pair<char,char> p=std::make_pair('x', 'y'); // пара двух символов

char c;
std::tie(std::ignore,c) = p; // извлекаем второе значение в переменную c
                             // (игнорируя первое значение)
```

Фактически здесь пара `p` присваивается кортежу, в котором второе значение является ссылкой на переменную `c` (см. раздел 5.1.2).

Сравнение пар

Для сравнения двух пар стандартная библиотека языка C++ предусматривает обычные операторы сравнения. Две пары считаются равными, если попарно равны оба значения пар.

```
namespace std {
   template <typename T1, typename T2>
   bool operator== (const pair<T1,T2>& x, const pair<T1,T2>& y) {
      return x.first == y.first && x.second == y.second;
   }
}
```

При сравнении пар больший приоритет имеет первое значение. Таким образом, если первые значения двух пар отличаются друг от друга, то результат их сравнения является результатом сравнения пар в целом. Если члены `first` равны, результатом сравнения пар является результат сравнения элементов `second`.

```
namespace std {
   template <typename T1, typename T2>
   bool operator< (const pair<T1,T2>& x, const pair<T1,T2>& y) {
      return x.first < y.first ||
            (!(y.first < x.first) && x.second < y.second);
   }
}
```

Остальные операторы сравнения определены аналогично.

Примеры использования пар

Стандартная библиотека C++ интенсивно использует пары. Например, (неупорядоченные) отображения и мультиотображения для управления своими элементами используют объекты типа `pair`, состоящие из ключа и значения. Общее описание этих контейнеров приведено в разделе 7.8, а пример использования типа `pair` приведен, в частности, в разделе 6.2.2.

Объекты типа `pair` используются также в стандартных библиотечных функциях языка C++, возвращающих два значения (например, см. раздел 7.7.2).

5.1.2. Кортежи

Кортежи появились в документе TR1 для расширения концепции пар на произвольное количество элементов. Иначе говоря, кортежи представляют собой неоднородный список элементов, типы которых задаются или выводятся во время компиляции.

Однако в документе TR1 использовались языковые возможности стандарта C++98, поэтому было невозможно определить шаблон для переменного количества элементов. По этой причине в реализациях необходимо было указывать все возможные количества элементов, которые могут содержать кортежи. В документации TR1 рекомендовалось поддерживать не меньше 10 аргументов. Это означало, что кортежи обычно определялись следующим образом (хотя некоторые реализации могли обеспечивать большее количество шаблонных параметров):

```
template <typename T0 = ..., typename T1 = ..., typename T2 = ...,
          typename T3 = ..., typename T4 = ..., typename T5 = ...,
          typename T6 = ..., typename T7 = ..., typename T8 = ...,
          typename T9 = ...>
class tuple;
```

Иначе говоря, класс `tuple` имел не меньше 10 параметров разных типов, а реализации по умолчанию назначали неиспользуемым элементам тип, не имеющий функциональных свойств. По сути, это была эмуляция вариативных шаблонов, которая на практике была довольно громоздкой и очень ограниченной.

В стандарте C++11 появились вариативные шаблоны, позволяющие шаблонам получать произвольное количество шаблонных аргументов (см. раздел 3.1.9). Вследствие этого объявление класса `tuple` в заголовке `<tuple>` сократилось.

```
namespace std {
   template <typename... Types>
   class tuple;
}
```

Операции над кортежами

В принципе интерфейс кортежа очень прост.

- Можно создать кортеж, либо явно объявив его, либо неявно с помощью вспомогательной функции `make_tuple()`.
- Можно получить доступ к элементам с помощью шаблонной функции `get<>()`.

Рассмотрим типичный пример этого интерфейса:

```
// util/tuple1.cpp

#include <tuple>
#include <iostream>
#include <complex>
#include <string>
using namespace std;

int main()
{
```

```
    // создаем кортеж из четырех элементов
    // - элементы инициализируются значением, заданным по умолчанию
    // (0 для элементарных типов)
    tuple<string,int,int,complex<double>> t;

    // явно создаем и инициализируем кортеж
    tuple<int,float,string> t1(41,6.3,"nico");

    // "проходим" по элементам:
    cout << get<0>(t1) << " ";
    cout << get<1>(t1) << " ";
    cout << get<2>(t1) << " ";
    cout << endl;

    // создаем кортеж с помощью функции make_tuple()
    // - ключевое слово auto объявляет объект t2 с типом правой части
    // - т.е. объект t2 имеет тип tuple
    auto t2 = make_tuple(22,44,"nico");

    // присваиваем второе значение кортежа t2 кортежу t1
    get<1>(t1) = get<1>(t2);

    // сравнение и присвоение
    // включая преобразование типа из tuple<int,int,const char*>
    // в tuple<int,float,string>
    if (t1 < t2) {  // поочередно сравниваем значения
        t1 = t2;    // OK, присваиваем значения
    }
}
```

Следующий оператор создает неоднородный кортеж из четырех элементов:

```
tuple<string,int,int,complex<double>> t;
```

Значения инициализируются своими конструкторами, заданными по умолчанию. Объекты элементарных типов инициализируются нулем (это гарантируется только в стандарте C++11).

Оператор

```
tuple<int,float,string> t1(41,6.3,"nico");
```

создаст и инициализирует неоднородный кортеж из трех элементов.

В качестве альтернативы можно использовать функцию `make_tuple()`, чтобы создать кортеж, в котором типы автоматически выводятся из начальных значений. Например, в следующем примере создается и инициализируется кортеж, состоящий из объектов, имеющих типы `int`, `int`, и `const char*`:[7]

```
make_tuple(22,44,"nico")
```

Отметим, что тип элемента кортежа может быть ссылкой. Например:

[7] Строка "nico" имеет тип `const char[5]`, но он "*ослабляется*" до `const char*` с помощью свойства `std::decay()` (см. раздел 5.4.2).

```
string s;
tuple<string&> t(s); // первый элемента кортежа t ссылается на s

get<0>(t) = "hello"; // присваиваем "hello" строке s
```

Кортеж не является обычным контейнерным классом, в котором можно осуществлять обход элементов. Вместо этого для доступа к элементам используются шаблонные члены, поэтому для доступа к элементам во время компиляции необходимо знать их индексы. Например, чтобы получить доступ к первому элементу кортежа `t1`, следует выполнить оператор

```
get<0>(t1)
```

Передача индекса во время выполнения программы невозможна.

```
int i;
get<i>(t1) // ошибка компиляции: i - не является значением времени компиляции
```

Хорошей новостью является то, что при передаче неправильного индекса возникает ошибка компиляции.

```
get<3>(t1) // ошибка компиляции, если кортеж t1 содержит только три элемента
```

Кроме того, кортежи предусматривают операции для обычного копирования, присваивания и сравнения. Для этих операций возможно неявное преобразование типа (поскольку используются шаблонные члены), но количество элементов должно совпадать. Кортежи считаются равными. если все их соответствующие элементы равны между собой. Для проверки, является ли кортеж меньше другого кортежа, применяется лексикографическое сравнение (см. раздел 11.5.4).

Операции над кортежами перечислены в табл. 5.2.

Таблица 5.2. Операции над кортежами

Операция	Действие
`tuple<T1,T2,...,Tn> t`	Создает кортеж из *n* элементов указанных типов, инициализированных своими конструкторами, заданными по умолчанию (0 для элементарных типов)
`tuple<T1,T2,...,Tn> t(v1,v2,...,vn)`	Создает кортеж из *n* элементов указанных типов, инициализированных заданными значениями
`tuple<T1,T2> t(p)`	Создает кортеж из двух элементов указанных типов, инициализированных значениями, переданными в объекте *p* типа `pair` (типы элементов в объекте *p* должны соответствовать указанным)
t = *t2*	Присваивает значения кортежа *t2* кортежу *t*
t = *p*	Присваивает объект *p* типа `pair` кортежу, состоящему из двух элементов (типы элементов в объекте *p* должны соответствовать указанным)

Окончание табл. 5.2

Операция	Действие
t1 == *t2*	Возвращает результат сравнения на равенство кортежей *t1* и *t2* (`true`, если сравнение с помощью оператора == для всех элементов возвращает значение `true`)
t1 != *t2*	Возвращает результат сравнения на неравенство кортежей *t1* и *t2* (! (*t1*==*t2*))
t1 < *t2*	Возвращает результат проверки, является ли кортеж *t1* меньшим кортежа *t2* (на основе лексикографического сравнения)
t1 > *t2*	Возвращает результат проверки, является ли кортеж *t1* бо́льшим кортежа *t2* (*t2*<*t1*)
t1 <= *t2*	Возвращает результат проверки, не превышает ли кортеж *t1* кортеж *t2* (! (*t2*<*t1*))
t1 >= *t2*	Возвращает результат проверки, не превышает ли кортеж *t2* кортеж *t1* (! (*t2*<*t1*)) (! (*t1*<*t2*))
t1.`swap(`*t2*`)`	Обменивает данные кортежей *t1* и *t2* (по стандарту C++11)
`swap(`*t1*`,`*t2*`)`	То же самое (с помощью глобальной функции) (по стандарту C++11)
`make_tuple(`*v1*`,`*v2*`,...)`	Создает кортеж с типами и значениями всех передаваемых значений и позволяет извлекать значения из кортежа
`tie(`*ref1*`,`*ref2*`,...)`	Создает кортеж ссылок, позволяя извлекать (отдельные) значения из кортежа

Вспомогательные функции `make_tuple()` и `tie()`

Вспомогательная функция `make_tuple()` создает кортеж значений без явного указания их типов. Например, выражение

```
make_tuple(22,44,"nico")
```

создает и инициализирует кортеж объектов соответствующих типов `int`, `int` и `const char*`.

Используя специальный функциональный объект `reference_wrapper<>` и его вспомогательные функции `ref()` и `cref()` (доступные в заголовке `<functional>` в соответствии со стандартом C++11; см. раздел 5.4.3), можно влиять на тип объекта, создаваемого функцией `make_tuple()`. Например, следующее выражение создает кортеж ссылок на переменную/объект `x`:

```
string s;

make_tuple(ref(s)) // создает объект типа tuple<string&>,
                   // в котором элемент ссылается на s
```

Это может быть важным, если вы хотите модифицировать существующее значение с помощью кортежа.

```
std::string s;

auto x = std::make_tuple(s);       // x — объект типа tuple<string>
std::get<0>(x) = "my value";       // модифицируем x, но не s

auto y = std::make_tuple(ref(s)); // y — объект типа tuple<string&>,
                                  // следовательно, y ссылается на s
std::get<0>(y) = "my value";       // модифицируем с помощью объекта y
```

Используя ссылки в функции `make_tuple()`, можно извлекать значения из кортежа и передавать их другим переменным. Рассмотрим следующий пример:

```
std::tuple <int,float,std::string> t(77,1.1,"more light");
int i;
float f;
std::string s;
// присваиваем значения кортежа t переменным i, f и s:
std::make_tuple(std::ref(i),std::ref(f),std::ref(s)) = t;
```

Для того чтобы сделать работу со ссылками в кортеже еще более удобной, функция `tie()` создает кортеж ссылок.

```
std::tuple <int,float,std::string> t(77,1.1,"more light");
int i;
float f;
std::string s;
std::tie(i,f,s) = t; // присваиваем значения кортежа t
                     // переменным i, f и s
```

Здесь вызов функции `std::tie(i,f,s)` создает кортеж ссылок на переменные `i`, `f` и `s`, поэтому присваивание объекта `t` означает присваивание его элементов переменным `i`, `f` и `s`.

Использование параметра `std::ignore` позволяет игнорировать элементы кортежа при его анализе с помощью функции `tie()`. Это позволяет избирательно извлекать значения из кортежа.

```
std::tuple <int,float,std::string> t(77,1.1,"more light");
int i;
std::string s;
std::tie(i,std::ignore,s) = t; // присваиваем первое и третье значение
                               // кортежа t переменным i и s
```

Кортежи и списки инициализации

Конструктор, получающий переменное количество аргументов для инициализации кортежа, объявляется с ключевым словом `explicit`.

```
namespace std {
   template <typename... Types>
   class tuple {
```

```
    public:
      explicit tuple(const Types&...);
      template <typename... UTypes> explicit tuple(UTypes&&...);
      ...
  };
}
```

Это позволяет избежать неявного преобразования отдельных значений из кортежей, содержащих только один элемент.

```
template <typename... Args>
void foo (const std::tuple<Args...> t);

foo(42);              // ОШИБКА: требуется явное
                      // преобразование в тип tuple<>
foo(make_tuple(42)); // OK
```

Однако эта ситуация имеет последствия при использовании списков инициализации для определения значений кортежа. Например, для инициализации кортежа нельзя использовать синтаксис присваивания, потому что оно будет рассматриваться как неявное преобразование.

```
std::tuple<int,double> t1(42,3.14);     // OK, старый синтаксис
std::tuple<int,double> t2{42,3.14};     // OK, новый синтаксис
std::tuple<int,double> t3 = {42,3.14}; // ОШИБКА
```

Кроме того, нельзя передавать список инициализации там, где ожидается кортеж.

```
std::vector<std::tuple<int,float>> v { {1,1.0}, {2,2.0} }; // ОШИБКА

std::tuple<int,int,int> foo() {
  return { 1, 2, 3 }; // ОШИБКА
}
```

Однако заметим, что этот метод работает в случае объектов типа `pair<>` и контейнеров (за исключением объектов типа `array<>`).

```
std::vector<std::pair<int,float>> v1 { {1,1.0}, {2,2.0} }; // OK
std::vector<std::vector<float>>   v2 { {1,1.0}, {2,2.0} }; // OK

std::vector<int> foo2() {
  return { 1, 2, 3 }; // OK
}
```

Но при работе с кортежами необходимо явно преобразовывать начальные значения в кортеж (например, с помощью функции `make_tuple()`).

```
std::vector<std::tuple<int,float>> v { std::make_tuple(1,1.0),
                                       std::make_tuple(2,2.0) }; // OK

std::tuple<int,int,int> foo() {
   return std::make_tuple(1,2,3); // OK
}
```

Дополнительные возможности кортежей

Для кортежей объявлено несколько вспомогательных средств, в частности, для поддержки обобщенного программирования.

- Член `tuple_size<`*тип_кортежа*`>::value` содержит количество элементов.
- Член `tuple_element<`*idx*`,`*тип_кортежа*`>::type` содержит информацию о типе элемента с индексом *idx* (т.е. информацию о типе значения, возвращаемого функцией `get()`).
- Функция `tuple_cat()` конкатенирует несколько кортежей в один кортеж.

Использование классов `tuple_size<>` и `tuple_element<>` демонстрируется в следующем примере:

```
typedef std::tuple<int,float,std::string> TupleType;

std::tuple_size<TupleType>::value       // выдает 3
std::tuple_element<1,TupleType>::type   // выдает float
```

Функцию `tuple_cat()` можно использовать для конкатенации любых кортежей, включая `pair<>`.

```
int n;
auto tt = std::tuple_cat (std::make_tuple(42,7.7,"hello"),
                          std::tie(n));
```

Здесь объект `tt` становится кортежем, содержащим элементы всех передаваемых кортежей, с учетом того, что последний элемент является ссылкой на переменную `n`.

5.1.3. Ввод-вывод кортежей

Класс `tuple` впервые появился в библиотеке Boost (см. [Boost]). Там класс `tuple` имел интерфейс для записи значений в потоки вывода, но в стандартной библиотеке C++ этой возможности не было. Используя следующий заголовочный файл, с помощью стандартного оператора вывода << можно выводить на экран любой кортеж[8].

```
// util/printtuple.hpp

#include <tuple>
#include <iostream>

// вспомогательная функция: выводит элементы, начиная с индекса IDX,
// из кортежа, содержащего MAX элементов
template <int IDX, int MAX, typename... Args>
struct PRINT_TUPLE {
   static void print (std::ostream& strm, const std::tuple<Args...>& t) {
      strm << std::get<IDX>(t) << (IDX+1==MAX ? "" : ",");
      PRINT_TUPLE<IDX+1,MAX,Args...>::print(strm,t);
```

[8] Отметим, что этот оператор вывода не работает без механизма *ADL* (*argument-dependent lookup — поиск с учетом аргументов*) (см. раздел 15.11.1).

```
    }
};

// частичная специализация для завершения рекурсии
template <int MAX, typename... Args>
struct PRINT_TUPLE<MAX,MAX,Args...> {
    static void print (std::ostream& strm, const std::tuple<Args...>& t) {
    }
};

// оператор вывода для кортежей
template <typename... Args>
std::ostream& operator << (std::ostream& strm,
                           const std::tuple<Args...>& t)
{
    strm << "[";
    PRINT_TUPLE<0,sizeof...(Args),Args...>::print(strm,t);
    return strm << "]";
}
```

Этот код для рекурсивного обхода элементов кортежа во время компиляции интенсивно использует шаблонное метапрограммирование. При каждом вызове функция `PRINT_TUPLE<>::print()` выводит один элемент и рекурсивно вызывает ту же самую функцию для следующего элемента. Частичная специализация, в которой текущий индекс `IDX` и количество элементов в кортеже `MAX` равны между собой, завершает рекурсию. Например, программа

```
// util/tuple2.cpp
#include "printtuple.hpp"
#include <tuple>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    tuple <int,float,string> t(77,1.1,"more light");
    cout << "io: " << t << endl;
}
```

выводит на экран следующий результат:

```
io: [77,1.1,more light]
```

Здесь оператор вывода

```
cout << t
```

вызывает функцию

```
PRINT_TUPLE<0,3,Args...>::print(cout,t);
```

5.1.4. Преобразования типов tuples и pairs

Как показано в табл. 5.2, двухэлементный кортеж можно инициализировать объектом типа `pair`. Кроме того, можно присваивать объект типа `pair` двухэлементному кортежу.

Отметим, что класс `pair<>` содержит специальный конструктор для инициализации его элементов с помощью кортежей. Детали описаны в разделе 5.1.1. Кроме того, другие типы тоже можно снабдить интерфейсом, напоминающим интерфейс кортежа. На самом деле в классах `pair<>` (см. раздел 5.1.1) и `array<>` (см. раздел 7.2.5) так и сделано.

5.2. Интеллектуальные указатели

Еще со времен языка C все знают, что указатели играют важную роль, но одновременно являются источником беспокойства. Одна из причин использования указателей — желание использовать семантику ссылок за пределами обычной области видимости. Однако иногда очень сложно гарантировать, что время жизни указателей и время жизни объектов, на которые они ссылаются, совпадают, особенно если на один и тот же объект ссылаются несколько указателей. Например, для того чтобы хранить один и тот же объект в нескольких коллекциях (см. главу 7), необходимо передать указатель в каждую коллекцию, и в идеале при уничтожении одного указателя никаких проблем быть не должно (не возникнут "висячие указатели" и не будет многократных удалений объекта, на которые ссылался удаленный указатель). Аналогично не должно быть проблем при удалении последней ссылки на объект ("утечка памяти" не возникнет).

Обычно для предотвращения таких проблем используются "интеллектуальные указатели". Они являются интеллектуальными в том смысле, что позволяют избежать проблем, описанных выше. Например, указатель может быть настолько интеллектуальным, чтобы "знать", является ли он последним указателем, ссылающимся на объект, и использовать это знание для уничтожения соответствующего объекта только при удалении "последнего владельца" этого объекта.

Отметим, что одного класса интеллектуальных указателей недостаточно. Интеллектуальные указатели могут учитывать разные аспекты и обладать разными свойствами, поскольку программисты готовы нести дополнительные затраты для обеспечения их интеллекта. Следует подчеркнуть, что ни один конкретный интеллектуальный указатель все же не исключает неправильное использование указателей и не предотвращает ошибочную работу программы.

В соответствии со стандартом C++11 стандартная библиотека языка C++ содержит два типа интеллектуальных указателей.

1. Класс `shared_ptr` предназначен для работы с указателями, реализующими концепцию *совместного владения* (shared ownership). Несколько интеллектуальных указателей могут ссылаться на один и тот же объект, так что объект и его ресурсы будут освобождены, когда последняя ссылка на него будет уничтожена. Для решения этой задачи в рамках более сложных сценариев предусмотрены вспомогательные классы, в частности `weak_ptr`, `bad_weak_ptr` и `enable_shared_from_this`.
2. Класс `unique_ptr` предназначен для работы с указателями, реализующими концепцию *эксклюзивного, или строгого владения* (exclusive ownership or strict ownership). Такие указатели гарантируют, что в каждый момент времени на отдельный объект может ссылаться только один интеллектуальный указатель. Однако

владение можно передавать. Этот указатель особенно полезен для предотвращения утечки ресурсов, например, при пропуске оператора `delete` после создания объекта с помощью оператора `new` и генерирования исключения.

В соответствии со стандартом C++98 в стандартной библиотеке языка C++ существовал только один класс интеллектуальных указателей — класс `auto_ptr<>`, разработанный для решения задач, которые теперь решает класс `unique_ptr`. Однако из-за отсутствия языковых средств, таких как семантика перемещения для конструкторов и операторов присваивания, а также наличия других недостатков, этот класс оказался плохо понятным и уязвимым для ошибок. По этой причине после появления класса `shared_ptr` в документе TR1 и класса `unique_ptr` в стандарте C++11 класс `auto_ptr` был официально объявлен устаревшим. Это значит, что его следует использовать только при компиляции старых программ.

Все классы интеллектуальных указателей объявлены в заголовочном файле `<memory>`.

5.2.1. Класс shared_ptr

Почти все нетривиальные программы должны уметь одновременно обращаться к объектам из разных точек кода. Таким образом, возникает необходимость "ссылаться" на объект из разных мест программы. Несмотря на то что язык предоставляет в распоряжение программиста ссылки и указатели, этого недостаточно, поскольку часто необходимо гарантировать, что при удалении последней ссылки на объект должен удаляться и сам объект, что может сопровождаться операциями по очистке памяти, например освобождения памяти или ресурсов.

Итак, нам нужна семантика "очистки, если объект больше не используется". Класс `shared_ptr` реализует семантику *совместного владения*. Следовательно, несколько совместно используемых указателей (shared pointers) могут совместно ссылаться на один и тот же объект, т.е. владеть им. Последний владелец объекта обязан уничтожить его и освободить связанные с ним ресурсы.

По умолчанию очистка производится с помощью выполнения оператора `delete`, при условии, что объект был создан с помощью оператора `new`. В то же время программист может (а часто просто обязан) определить другие способы очистки объектов. Программист может определить свою собственную *стратегию уничтожения* (destruction policy). Например, если объекты содержатся в массиве, размещенном в памяти с помощью оператора `new[]`, необходимо определить очистку, выполняющую оператор `delete[]`. Другие примеры удаления связанных ресурсов предусматривают освобождение связанных с объектом ресурсов, например дескрипторов, блокировок, временных файлов и т.д..

Итак, цель класса `shared_ptr` — автоматическое освобождение ресурсов, связанных с объектами, которые больше не нужны (но не раньше).

Использование класса shared_ptr

Объекты класса `shared_ptr` можно использовать как любой другой указатель. Таким образом, совместно используемые указатели можно присваивать, копировать и сравнивать, а также можно применять к ним операторы * и -> для доступа к объектам, на которые они ссылаются.

Рассмотрим следующий пример:

```
// util/sharedptr1.cpp

#include <iostream>
#include <string>
#include <vector>
#include <memory>
using namespace std;

int main()
{
   // два совместно используемых указателя,
   // представляющие двух людей по их именам
   shared_ptr<string> pNico(new string("nico"));
   shared_ptr<string> pJutta(new string("jutta"));

   // переводим имя в верхний регистр
   (*pNico)[0] = 'N';
   pJutta->replace(0,1,"J");

   // несколько раз записываем указатели в контейнер
   vector<shared_ptr<string>> whoMadeCoffee;
   whoMadeCoffee.push_back(pJutta);
   whoMadeCoffee.push_back(pJutta);
   whoMadeCoffee.push_back(pNico);
   whoMadeCoffee.push_back(pJutta);
   whoMadeCoffee.push_back(pNico);

  // Выводим на экран все элементы
   for (auto ptr : whoMadeCoffee) {
      cout << *ptr << " ";
   }
   cout << endl;

   // Перезаписываем имя
   *pNico = "Nicolai";

   // снова выводим на экран все элементы
   for (auto ptr : whoMadeCoffee) {
        cout << *ptr << " ";
   }
   cout << endl;

   // выводим на экран внутренние данные
   cout << "use_count: " << whoMadeCoffee[0].use_count() << endl;
}
```

После включения заголовка <memory>, в котором определен класс shared_ptr, объявляются и инициализируются два совместно используемых указателя.

```
shared_ptr<string> pNico(new string("nico"));
shared_ptr<string> pJutta(new string("jutta"));
```

Отметим, что поскольку конструктор, получающий указатель в качестве единственного аргумента, объявлен с ключевым словом `explicit`, присваивание в этом месте использовать нельзя, потому что оно интерпретируется как неявное преобразование. Однако можно использовать новый синтаксис инициализации.

```
shared_ptr<string> pNico = new string("nico"); // ОШИБКА
shared_ptr<string> pNico{new string("nico")};  // OK
```

Можно также использовать вспомогательную функцию `make_shared()`.

```
shared_ptr<string> pNico = make_shared<string>("nico");
shared_ptr<string> pJutta = make_shared<string>("jutta");
```

Этот способ создания указателя является более быстрым и безопасным, потому что он использует только одну операцию выделения памяти, а не две: одну для объекта, а другую — для совместно используемых данных, которые совместно используемый указатель использует для управления объектом (детали изложены в разделе 5.2.4).

В качестве альтернативы можно сначала объявить совместно используемый указатель, а затем присвоить ему новый. Однако оператор присваивания использовать нельзя; вместо него следует использовать функцию `reset()`.

```
shared_ptr<string> pNico4;
pNico4 = new string("nico");       // ОШИБКА: нельзя присваивать
                                   // обычные указатели

pNico4.reset(new string("nico")); // OK
```

Следующие две строки демонстрируют использование совместно используемых указателей как обычных указателей:

```
(*pNico)[0] = 'N';
pJutta->replace(0,1,"J");
```

После выполнения оператора * объект `pNico` ссылается на объект, в котором первому символу присваивается символ `'N'`. С помощью оператора -> мы получаем доступ к члену объекта `pJutta`. Таким образом, здесь функция-член `replace()` позволяет заменить подстроки (см. раздел 13.3.7).

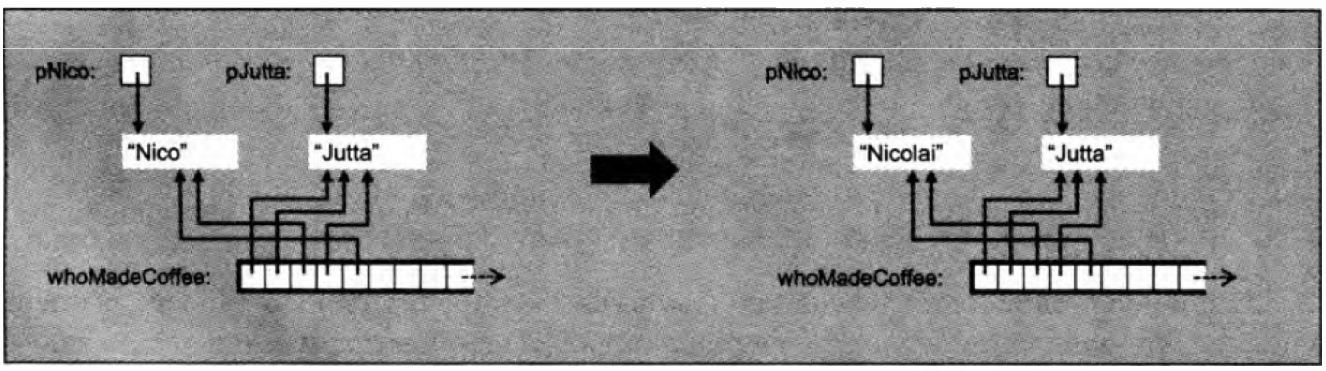


Рис. 5.1. Использование объектов класса `shared_ptr`

Далее мы несколько раз вставляем оба указателя в контейнер типа vector<> (см. раздел 7.3). Этот контейнер обычно создает свои копии передаваемых ему элементов, поэтому если бы вставляли строки непосредственно, то на самом деле вставляли бы их копии. Однако, поскольку мы передаем указатели на строки, эти указатели копируются и контейнер теперь содержит несколько ссылок на один и тот же объект. Это значит, что при попытке модифицировать этот объект будут изменены все его экземпляры, содержащиеся в контейнере.

Таким образом, заменив значение строки pNico

```
*pNico = "Nicolai";
```

мы вынуждаем все экземпляры этого объекта ссылаться на новое значение, как показано на рис. 5.1, а результат работы программы будет таким:

```
Jutta  Jutta  Nico  Jutta  Nico
Jutta  Jutta  Nicolai  Jutta  Nicolai
use_count: 4
```

Последняя строка вывода представляет собой результат вызова функции use_count() для первого совместно используемого указателя в векторе. Функция use_count() выдает текущее количество владельцев объекта, на который ссылаются совместно используемые указатели. Как можно убедиться, у объекта, на который ссылается первый элемент вектора, есть четыре владельца: pJutta и три его копии, вставленные в контейнер.

В конце программы, когда уничтожается последний владелец строки, совместно используемый указатель применяет оператор delete к объекту, на который он ссылается. Такое удаление не всегда должно выполняться именно на границе области видимости. Например, присвоение указателя nullptr (см. раздел 3.1.1) совместно используемому указателю pNico или изменение размера вектора, чтобы он содержал только два первых элемента, приведет к удалению последнего владельца строки "nico".

Определение оператора удаления

Программист может определить свой собственный оператор удаления, которая, например, выводит сообщение перед удалением объекта, на который ссылается совместно используемый указатель.

```
shared_ptr<string> pNico(new string("nico"),
                         [](string* p) {
                            cout << "delete " << *p << endl;
                            delete p;
                         });
...
pNico = nullptr;         // pNico больше не ссылается на строку
whoMadeCoffee.resize(2); // все копии строки в pNico уничтожены
```

Здесь мы передаем лямбда-функцию (см. раздел 3.1.10, стр. 28) как второй аргумент конструктора класса shared_ptr. Если указатель pNico объявлен именно так, то после удаления последнего владельца строки будет вызвана лямбда-функция. Таким образом, если модифицировать предыдущую программу так, как показано выше, то при вызове функции resize() после всех операторов, рассмотренных ранее, она выведет на экран

строку delete Nicolai. Тот же самый эффект возникнет, если сначала изменить размер вектора, а затем присвоить указателю pNico указатель nullptr или другой объект.

Другой пример приложения указателя класса shared_ptr<> демонстрирует, как элементы могут находиться в двух контейнерах одновременно (см. раздел 7.11).

Работа с массивами

Отметим, что оператор удаления, предусмотренный по умолчанию в классе shared_ptr, выполняет операцию delete, а не delete[]. Это означает, что оператор удаления, предусмотренный по умолчанию, является приемлемым, только если совместно используемый указатель владеет одним объектом, созданным с помощью оператора new.

К сожалению, создание объекта класса shared_ptr для массива возможно, но считается ошибкой.

```
std::shared_ptr<int> p(new int[10]); // ОШИБКА, хотя и компилируется
```

Итак, если вы используете оператор new[] для создания массива объектов, то должны предусмотреть свою собственную операцию удаления. Это можно сделать, передав функцию, функцию-объект или лямбда-функцию, выполняющую оператор delete[ ] для обычных указателей, передаваемых ей как аргументы. Рассмотрим пример:

```
std::shared_ptr<int> p(new int[10],
                       [](int* p) {
                          delete[] p;
                       });
```

Кроме того, можно использовать вспомогательную функцию, официально предоставляемую классом unique_ptr, которая выполняет оператор delete[] (см. раздел 5.2.5).

```
std::shared_ptr<int> p(new int[10],
                        std::default_delete<int[]>());
```

Отметим, однако, что указатели классов shared_ptr и unique_ptr имеют немного разные механизмы удаления. Например, указатели класса unique_ptr позволяют владеть массивом, просто передавая соответствующий тип элементов в качестве шаблонного аргумента, в то время как указатели класса shared_ptr этого делать не могут.

```
std::unique_ptr<int[]> p(new int[10]); // ОК
std::shared_ptr<int[]> p(new int[10]); // ОШИБКА: не компилируется
```

Кроме того, чтобы задать свой собственный оператор удаления для указателей класса unique_ptr, необходимо указывать второй шаблонный аргумент.

```
std::unique_ptr<int,void(*)(int*)> p(new int[10],
                                     [](int* p) {
                                        delete[] p;
                                     });
```

Отметим также, что класс shared_ptr не содержит оператор []. У класса unique_ptr есть частичная специализация для массивов, в которой вместо операторов * и -> предусмотрен оператор []. Причина таких различий заключается в том, что класс unique_ptr оптимизирован по быстродействию и гибкости. Детали изложены в разделе 5.2.8.

Реализация других стратегий удаления

Если очистка после удаления последнего совместно используемого указателя, владеющего объектом, не сводится к простому освобождению памяти, следует указать свой собственный оператор удаления, т.е. реализовать свою собственную *стратегию удаления* (destruction policy).

В качестве первого примера предположим, что требуется гарантировать удаление временных файлов при уничтожении последней ссылки. Вот как это можно сделать:

```
// util/sharedptr2.cpp
#include <string>
#include <fstream> // для потока ofstream
#include <memory>  // для класса shared_ptr
#include <cstdio>  // для функции remove()

class FileDeleter
{
   private:
      std::string filename;
   public:
      FileDeleter (const std::string& fn)
      : filename(fn) {
      }
      void operator () (std::ofstream* fp) {
         delete fp;                       // закрыть файл
         std::remove(filename.c_str()); // удалить файл
      }
};

int main()
{
   // создаем и открываем временный файл:
   std::shared_ptr<std::ofstream> fp(new std::ofstream("tmpfile.txt"),
                                     FileDeleter("tmpfile.txt"));
   ...
}
```

Здесь мы инициализируем объект класса `shared_ptr` созданным файлом вывода (см. раздел 15.9). Передаваемый объект класса `FileDeleter` гарантирует, что этот файл будет закрыт и удален с помощью стандартной функции `remove()` языка C, определенной в заголовке `<cstdio>`, когда последняя копия данного совместно используемого указателя потеряет владение данным потоком вывода. Поскольку функции `remove()` необходимо знать имя удаляемого файла, мы передаем его как аргумент конструктора класса `FileDeleter`.

Второй пример демонстрирует использование объектов класса `shared_ptr` для работы с совместно используемой памятью[9].

[9] Существует много способов работы с разделяемой памятью, зависящих от конкретных операционных систем. Здесь используется стандартный подход POSIX, основанный на вызове функций `shm_open()` и `mmap()`, которые в свою очередь требуют вызова функции `shm_unlink()` для освобождения (персистентной) разделяемой памяти.

```
// util/sharedptr3.cpp
#include <memory>      // для класса shared_ptr
#include <sys/mman.h>  // для совместно используемой памяти
#include <fcntl.h>
#include <unistd.h>
#include <cstring>     // для функции strerror()
#include <cerrno>      // для кода ошибки errno
#include <string>
#include <iostream>

class SharedMemoryDetacher
{
   public:
      void operator () (int* p) {
         std::cout << "unlink /tmp1234" << std::endl;
         if (shm_unlink("/tmp1234") != 0) {
             std::cerr << "OOPS: shm_unlink() failed" << std::endl;
         }
   }
};

std::shared_ptr<int> getSharedIntMemory (int num)
{
   void* mem;
   int shmfd = shm_open("/tmp1234", O_CREAT|O_RDWR, S_IRWXU|S_IRWXG);
   if (shmfd < 0) {
       throw std::string(strerror(errno));
   }
   if (ftruncate(shmfd, num*sizeof(int)) == -1) {
       throw std::string(strerror(errno));
   }
   mem = mmap(nullptr, num*sizeof(int), PROT_READ | PROT_WRITE,
              MAP_SHARED, shmfd, 0);
   if (mem == MAP_FAILED) {
       throw std::string(strerror(errno));
   }
   return std::shared_ptr<int>(static_cast<int*>(mem),
                               SharedMemoryDetacher());
   }

int main()
{
   // выделяем и связываем совместно используемую память
   // для 100 целых чисел:
   std::shared_ptr<int> smp(getSharedIntMemory(100));

   // инициализируем совместно используемую память
   for (int i=0; i<100; ++i) {
        smp.get()[i] = i*42;
   }

   // используем совместно используемую память:
   ...
```

```
    std::cout << "<return>" << std::endl;
    std::cin.get();

    // освобождаем совместно используемую память:
    smp.reset();
    ...
}
```

Сначала определяется объект класса `SharedMemoryDetacher`, освобождающий совместно используемую память, выделенную функцией `getSharedIntMemory()`. Для того чтобы гарантировать, что этот функциональный объект будет вызван после того, как совместно используемая память будет использована в последний раз, этот объект передается совместно используемому указателю, связанному с выделенной памятью, при его создании функцией `getSharedIntMemory()`.

```
return std::shared_ptr<int>(static_cast<int*>(mem),
                            SharedMemoryDetacher()); // вызывает shmdt()
```

В качестве альтернативы здесь можно использовать лямбда-функцию (пропускаем префикс `std::`).

```
return shared_ptr<int>(static_cast<int*>(mem),
                       [](int* p) {
                           cout << "unlink /tmp1234" << endl;
                           if (shm_unlink("/tmp1234") != 0) {
                               cerr << "OOPS: shm_unlink() failed"
                                    << endl;
                           }
                       });
```

Отметим, что передаваемый оператор удаления не может генерировать исключения. Следовательно, здесь можно только вывести в поток `std::cerr` сообщение об ошибке.

Поскольку сигнатура функции `shm_unlink()` уже соответствует сигнатуре "удалителя", мы можем просто использовать функцию `shm_unlink()` непосредственно как оператор удаления, если не хотим проверять ее возвращаемое значение.

```
return std::shared_ptr<int>(static_cast<int*>(mem),shm_unlink);
```

Отметим, что указатели `shared_ptr` предусматривают только операторы * и ->. Арифметика указателей и оператор `[]` не поддерживаются. Таким образом, при доступе к памяти для реализации полноценной семантики указателей необходимо использовать функцию `get()`, возвращающую внутренний указатель, упакованный в указатель `shared_ptr`.

```
smp.get()[i] = i*42;
```

Альтернативный вызов функции `get()` выглядит следующим образом:

```
(&*smp)[i] = i*42;
```

В обоих примерах возможна еще более простая реализация: просто создайте новый класс, в котором конструктор выполняет инициализацию, а деструктор — очистку. Затем можно использовать совместно используемые указатели для управления объектами данного класса, созданными с помощью оператора `new`. Преимущество этого подхода

заключается в том, что он позволяет определить более интуитивный интерфейс, такой как оператор [] для объекта, представляющего совместно используемую память. Однако следует хорошенько подумать об операторах копирования и присваивания (если сомневаетесь, отключите их).

5.2.2. Класс weak_ptr

Основная причина использования shared_ptr — желание избежать забот о ресурсах, на которые ссылается указатель. Как указывалось выше, указатели shared_ptr обеспечивают автоматическое освобождение ресурсов, связанных с объектами, которые больше не нужны.

Однако при определенных обстоятельствах это поведение является невозможным или нежелательным.

- Одним из примеров являются циклические ссылки. Если два объекта ссылаются друг на друга с помощью совместно используемых указателей shared_ptr и вы хотите освободить объекты и связанные с ними ресурсы при условии, что на них больше никто не ссылается, указатель shared_ptr не освободит данные, потому что функция use_count() для каждого объекта будет равной 1. В этой ситуации можно использовать обычные указатели, но при этом придется взять на себя управление освобождением связанных ресурсов.
- Другой пример — ситуация, в которой вы хотите раздельно использовать объект, но не владеть им. Таким образом, возникает семантика, в которой время жизни ссылки на объект превышает время жизни самого объекта. В этом случае указатели класса shared_ptr никогда не освободят объект, а обычные указатели могут не заметить, что ссылаются на объект, которого уже нет. В итоге возникает риск обращения к несуществующим данным.

Обе проблемы решает класс weak_ptr, позволяющий раздельно использовать объект, но не владеть им. Этот класс требует создания совместно используемого указателя. Как только последний совместно используемый указатель, владеющий объектом, потеряет владение, слабый указатель автоматически станет пустым. Таким образом, помимо конструктора по умолчанию и копирующего конструктора, класс weak_ptr содержит только конструктор, получающий аргумент типа shared_ptr.

Нельзя непосредственно использовать операторы * и -> для доступа к объекту, на который ссылается указатель weak_ptr. Вместо этого следует создать соответствующий совместно используемый указатель. Это целесообразно по двум причинам.

1. Создание совместно используемого указателя на основе слабого указателя позволяет выполнить проверку связанного с ним объекта. Если объект больше не связан с ним, указатель генерирует исключение или создает пустой совместно используемый указатель (точная реакция зависит от реализации этой операции).
2. При работе со связанным объектом совместно используемый указатель невозможно удалить.

По этой причине класс weak_ptr содержит только небольшой набор операторов: для создания, копирования и присваивания слабого указателя и преобразования его в совместно используемый указатель или проверки, ссылается ли он на какой-либо объект.

Использование класса weak_ptr

Рассмотрим следующий пример:

```
// util/weakptr1.cpp
#include <iostream>
#include <string>
#include <vector>
#include <memory>
using namespace std;

class Person {
  public:
    string name;
    shared_ptr<Person> mother;
    shared_ptr<Person> father;
    vector<shared_ptr<Person>> kids;

    Person (const string& n,
            shared_ptr<Person> m = nullptr,
            shared_ptr<Person> f = nullptr)
    : name(n), mother(m), father(f) {
    }

    ~Person() {
      cout << "delete " << name << endl;
    }
};

shared_ptr<Person> initFamily (const string& name)
{
   shared_ptr<Person> mom(new Person(name+"'s mom"));
   shared_ptr<Person> dad(new Person(name+"'s dad"));
   shared_ptr<Person> kid(new Person(name,mom,dad));
   mom->kids.push_back(kid);
   dad->kids.push_back(kid);
   return kid;
}

int main()
{
   shared_ptr<Person> p = initFamily("nico");

   cout << "nico's family exists" << endl;
   cout << "- nico is shared " << p.use_count() << " times" << endl;
   cout << "- name of 1st kid of nico's mom: "
        << p->mother->kids[0]->name << endl;

   p = initFamily("jim");
   cout << "jim's family exists" << endl;
}
```

Здесь класс `Person` хранит имя, возможно, ссылки на другие объекты класса `Person`, в частности, на родителей (`mother` и `father`) и детей (представленных вектором; см. раздел 7.3).

Сначала функция `initFamily()` создает три объекта класса `Person`: `mom`, `dad` и `kid`, инициализированных соответствующими именами на основе переданных аргументов. Кроме того, объект `kid` инициализируется данными о родителях, а в объектах, описывающих обоих родителей, объект `kid` вставляется в список детей. В заключение функция `initFamily()` возвращает объект `kid`. На рис. 5.2 показана итоговая ситуация после вызова функции `initFamily()` и присваивание результата ее работы объекту `p`.

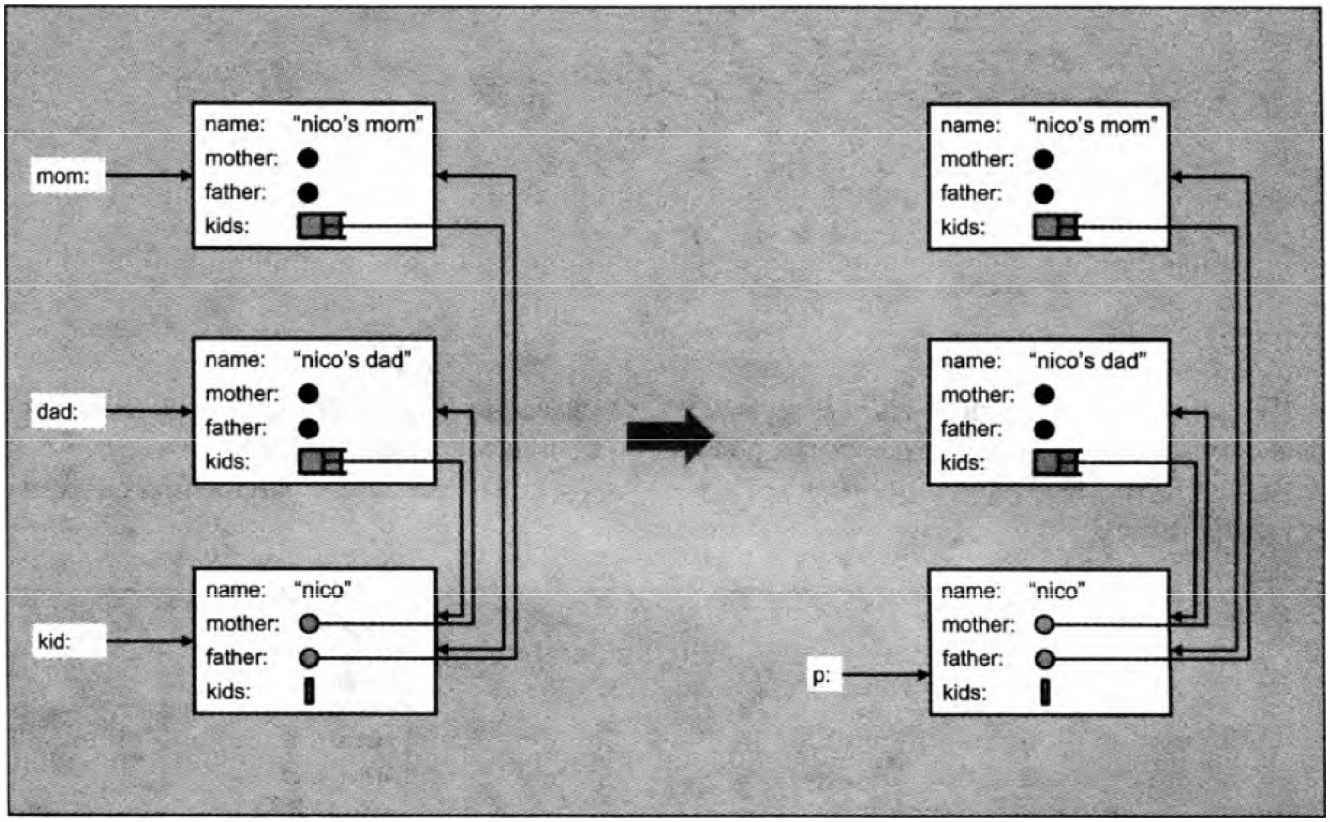


Рис. 5.2. Семейство, использующее только указатели `shared_ptr`

Как видим, объект `p` является последним указателем на созданное семейство. Однако каждый объект имеет ссылки от ребенка к каждому родителю, и наоборот. Например, на объект `nico` было установлено три ссылки, прежде чем указатель `p` получил новое значение. Теперь, если мы удалим последний указатель на семейство, присвоив ему либо новый объект, либо указатель `nullptr`, либо выйдя из области видимости указателя `p` в конце функции `main()`, ни один объект класса `Person` не будет удален из памяти, потому что будет существовать по крайней мере один совместно используемый указатель, ссылающийся на него. В результате деструктор каждого объекта класса `Person`, который должен был выводить сообщение "`delete` имя", никогда не будет вызван.

```
nico's family exists
- nico is shared 3 times
- name of 1st kid of nicos mom: nico
jim's family exists
```

Использование указателей класса `weak_ptr` решает эту проблему. Например, можно объявить вектор `kids` как содержащий указатели `weak_ptr`.

```
// util/weakptr2.cpp
...
class Person {
   public:
      string name;
      shared_ptr<Person> mother;
      shared_ptr<Person> father;
      vector<weak_ptr<Person>> kids;    // слабый указатель !!!
      Person (const string& n,
              shared_ptr<Person> m = nullptr,
              shared_ptr<Person> f = nullptr)
              : name(n), mother(m), father(f) {
   }
   ~Person() {
      cout << "delete " << name << endl;
   }
};
...
```

Теперь мы можем разорвать замкнутый круг совместно используемых указателей, чтобы в одном направлении (от ребенка к родителю) использовался совместно используемый указатель, а от родителя к детям использовались слабые указатели (пунктирная линия на рис. 5.3).

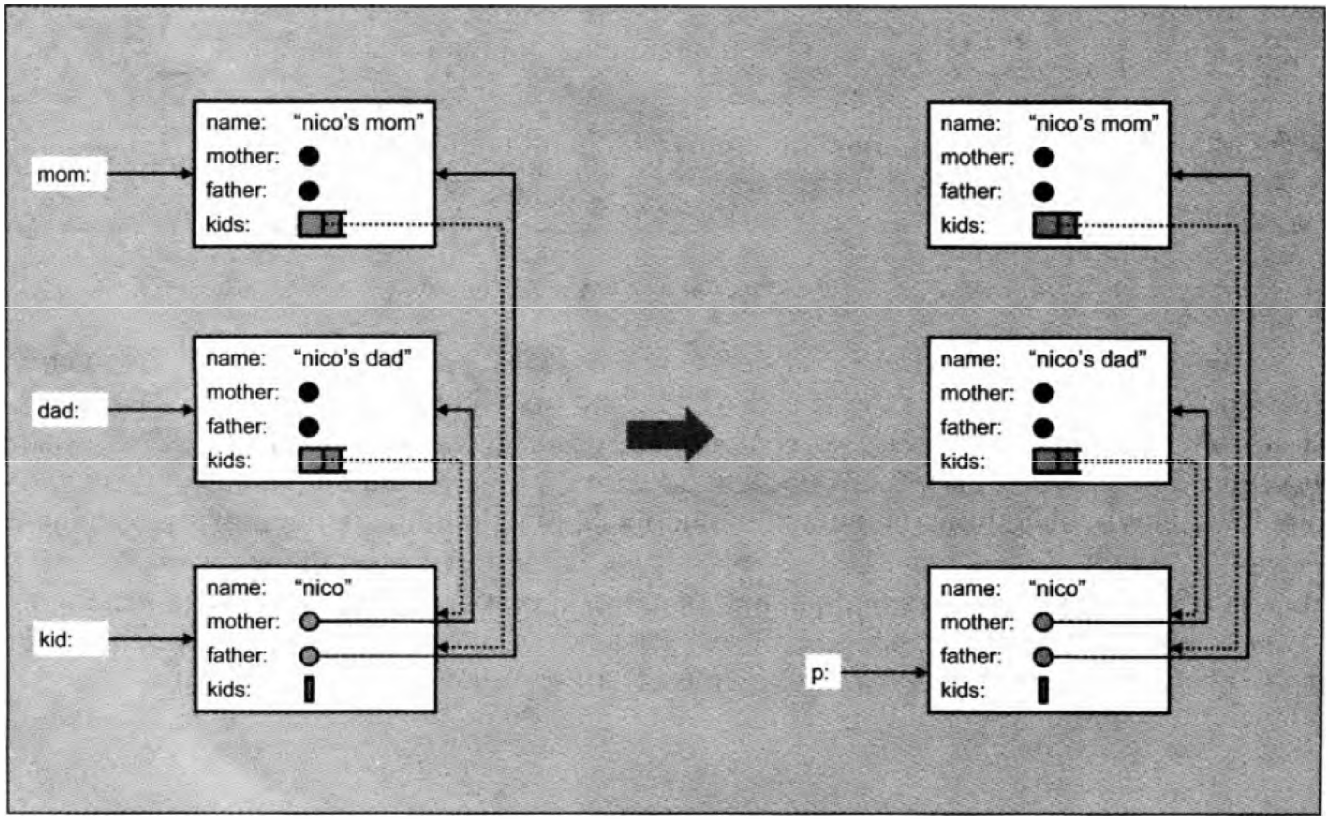


Рис. 5.3. Семейство, использующее указатели классов `shared_ptr` и `weak_ptr`

В результате программа будет выводить следующие результаты:

```
nico's family exists
- nico is shared 1 times
- name of 1st kid of nicos mom: nico
delete nico
delete nico's dad
delete nico's mom
jim's family exists
delete jim
delete jim's dad
delete jim's mom
```

Как только мы потеряем указатель на созданное семейство, либо присвоив указателю `p` новое значение, либо выйдя из функции `main()`, объект в массиве `kids` потеряет последнего владельца (перед этим функция `use_count()` выдаст значение `1`). В итоге оба объекта, описывающих родителей, потеряют своего последнего владельца. Итак, все объекты, созданные с помощью оператора `new`, теперь удаляются с помощью оператора `delete`, так что их деструкторы вызываются.

Отметим, что, используя слабые указатели, мы немного изменяем способ доступа к объекту, на который ссылается слабый указатель. Вместо вызова

```
p->mother->kids[0]->name
```

мы теперь должны вставить вызов функции `lock()` в выражение

```
p->mother->kids[0].lock()->name
```

В результате из указателей класса `weak_ptr`, содержащихся в векторе `kids`, будут созданы указатели класса `shared_ptr`. Если эта модификация невозможна, например, потому, что последний владелец объекта уже освободил его, то функция `lock()` вернет пустой указатель `shared_ptr`. В этом случае вызов оператора `*` или `->` приведет к непредсказуемым последствиям.

Если вы не уверены, что объект, на который ссылается слабый указатель, все еще существует, можно выбрать один из вариантов, описанных ниже.

1. Функцию `expired()`, которая возвращает значение `true`, если указатель `weak_ptr` больше не владеет объектом. Это эквивалентно проверке, равен ли нулю результат функции `use_count()`, но функция `expired()` работает быстрее.
2. Явно преобразовать указатель `weak_ptr` в указатель `shared_ptr`, используя соответствующий конструктор `shared_ptr`. Если объекта, на который ссылается указатель, не существует, этот конструктор сгенерирует исключение `bad_weak_ptr`. Это исключение является производным от класса `std::exception`, в котором функция `what()` выдает строку `"bad_weak_ptr"`.[10] Детальную информацию о стандартных исключениях см. в разделе 4.3.1.

[10] Для исключений значение, возвращаемое функцией `what()`, обычно зависит от реализации. Однако стандарт указывает, что в данном случае функция `what()` должна возвращать строку "`bad_weak_ptr`". Тем не менее реализации могут не следовать этому совету; например, компилятор GCC 4.6.1 возвращает строку "`std::bad_weak_ptr`".

3. Вызвать функцию use_count() и запросить количество владельцев, связанных с объектом. Если функция вернет значение 0, корректного объекта больше нет. Отметим, однако, что функцию use_count() следует вызывать только для отладки; стандарт языка C++ явно утверждает: "функция use_count() не всегда эффективна".

Например:

```
try {
   shared_ptr<string> sp(new string("hi"));  // создаем совместно
                                             // используемый указатель
   weak_ptr<string> wp = sp;                 // создаем из него слабый указатель
   sp.reset();                               // удаляем объект, на который
                                             // ссылается совместно
                                             // используемый указатель
   cout << wp.use_count() << endl;           // вывод: 0
   cout << boolalpha << wp.expired() << endl;// вывод: true
   shared_ptr<string> p(wp);                 // генерирует std::bad_weak_ptr
}
catch (const std::exception& e) {
   cerr << "exception: " << e.what() << endl;// вывод: bad_weak_ptr
}
```

5.2.3. Неправильное использование совместно используемых указателей

Несмотря на то что указатели shared_ptr повышают безопасность программы, поскольку общие ресурсы, связанные с объектами, освобождаются автоматически, если объекты больше не используются, могут возникнуть проблемы. Одна из вышеописанных проблем называется "висячие указатели" и вызывается циклическими зависимостями.

Другая проблема заключается в том, что программист должен гарантировать, что только одна группа совместно используемых указателей будет владеть объектом. Следующий код работать не будет:

```
int* p = new int;
shared_ptr<int> sp1(p);
shared_ptr<int> sp2(p); // ОШИБКА: переменной типа int, созданной в памяти,
                        // управляют два совместно используемых указателя
```

Проблема заключается в том, что оба указателя, sp1 и sp2, должны освобождать связанные с ними ресурсы (выполняя оператор delete), когда они теряют владение переменной p. В принципе, существование двух групп владельцев означает, что освобождение связанных ресурсов выполняется дважды, независимо оттого, потерял ли владение последний владелец каждой группы или был удален. По этой причине интеллектуальный указатель необходимо всегда инициализировать непосредственно в момент создания объекта с ресурсами, ассоциированными с ним.

```
shared_ptr<int> sp1(new int);
shared_ptr<int> sp2(sp1); // OK
```

Эта проблема может возникнуть и опосредованно. Предположим, что в только что рассмотренном примере мы хотим ввести в класс `Person` функцию-член, создающую ссылку от ребенка к родителю и соответствующую ссылку в обратном направлении.

```
shared_ptr<Person> mom(new Person(name+"'s mom"));
shared_ptr<Person> dad(new Person(name+"'s dad"));
shared_ptr<Person> kid(new Person(name));
kid->setParentsAndTheirKids(mom,dad);
```

Рассмотрим наивную реализацию функции `setParentsAndTheirKids()`.

```
class Person {
   public:
   ...
      void setParentsAndTheirKids (shared_ptr<Person> m = nullptr,
                                   shared_ptr<Person> f = nullptr) {
         mother = m;
         father = f;
         if (m != nullptr) {
             m->kids.push_back(shared_ptr<Person>(this)); // Ошибка
         }
         if (f != nullptr) {
             f->kids.push_back(shared_ptr<Person>(this)); // Ошибка
         }
   }
   ...
};
```

Наша цель — создать совместно используемый указатель из указателя `this`. Мы делаем это потому, что хотим задать вектор `kids` в членах классов `mother` и `father`. Однако для этого необходимо иметь совместно используемый указатель на ребенка, а этого указателя у нас нет. Создание же нового совместно используемого указателя из указателя `this` не решает проблему, потому что мы создаем новую группу владельцев.

Справиться с этой ситуацией могла бы передача совместно используемого указателя на ребенка в качестве третьего аргумента. Однако в стандартной библиотеке языка C++ есть другая возможность: класс `std::enable_shared_from_this<>`.

Из класса `std::enable_shared_from_this<>` можно создать производный класс, представляющий объекты, управляемые совместно используемыми указателями, и передать имя своего класса как шаблонный аргумент. Это позволяет использовать производную функцию-член `shared_from_this()` для корректного создания указателя `shared_ptr` на основе указателя `this`.

```
   class Person : public std::enable_shared_from_this<Person> {
   public:
      ...
      void setParentsAndTheirKids (shared_ptr<Person> m = nullptr,
                                   shared_ptr<Person> f = nullptr) {
          mother = m;
          father = f;
          if (m != nullptr) {
              m->kids.push_back(shared_from_this()); // OK
          }
```

```
        if (f != nullptr) {
            f->kids.push_back(shared_from_this()); // OK
        }
    }
    ...
};
```

Полностью программа записана в файле `util/enableshared1.cpp`.

Отметим, что функцию `shared_from_this()` нельзя вызывать в конструкторе (можно в смысле успешной компиляции, но в результате возникнет ошибка на этапе выполнения программы).

```
class Person : public std::enable_shared_from_this<Person> {
  public:
    ...
    Person (const string& n,
            shared_ptr<Person> m = nullptr,
            shared_ptr<Person> f = nullptr)
    : name(n), mother(m), father(f) {
      if (m != nullptr) {
          m->kids.push_back(shared_from_this()); // ОШИБКА
      }
      if (f != nullptr) {
          f->kids.push_back(shared_from_this()); // ОШИБКА
      }
  }
  ...
};
```

Проблема состоит в том, что указатель класса `shared_ptr` сохраняется в закрытом члене `enable_shared_from_this<>` базового класса `Person` в конце процедуры создания объекта класса `Person`.

Итак, абсолютно невозможно создать циклические ссылки совместно используемых указателей во время создания объекта, инициализирующего совместно используемый указатель. Это можно сделать в два этапа — тем или иным способом.

5.2.4. Подробное описание совместно используемых и слабых указателей

Подведем итоги и опишем полный интерфейс совместно используемых и слабых указателей.

Подробное описание класса `shared_ptr`

Как указано в разделе 5.2.1, класс `shared_ptr` реализует концепцию интеллектуального указателя с семантикой совместного владения. Всякий раз, когда уничтожается последний владелец совместного указателя, ассоциированный объект также удаляется (или освобождаются связанные с ним ресурсы).

Шаблонным параметром класса `shared_ptr<>` является тип объекта, которым инициализируется указатель.

```
namespace std {
   template <typename T>
   class shared_ptr
   {
      public:
        typedef T element_type;
        ...
   };
}
```

Тип элемента может быть `void`. Это значит, что совместно используемый указатель разделяет владение объектом неопределенного типа, аналогично указателям типа `void*`.

Пустой указатель типа `shared_ptr` не разделяет владение объектом, поэтому функция `use_count()` выдает значение `0`. Отметим, однако, что благодаря специальному конструктору совместно используемый указатель все еще может ссылаться на объект.

Все операции, предусмотренные для совместно используемых указателей, приведены в табл. 5.3.

Таблица 5.3. Операции над объектами класса `shared_ptr`

Операция	Действие
`shared_ptr<`*T*`>` *sp*	Конструктор по умолчанию; создает пустой совместно используемый указатель, используя механизм удаления, предусмотренный по умолчанию (оператор `delete`)
`shared_ptr<`*T*`>` *sp*(*ptr*)	Создает совместно используемый указатель, владеющий объектом **ptr*, используя механизм удаления, предусмотренный по умолчанию (оператор `delete`)
`shared_ptr<`*T*`>` *sp*(*ptr*, *del*)	Создает совместно используемый указатель, владеющий объектом **ptr*, используя функцию *del* в качестве оператора удаления
`shared_ptr<`*T*`>` *sp*(*ptr*, *del*, *ac*)	Создает совместно используемый указатель, владеющий объектом **ptr*, используя функцию *del* в качестве оператора удаления и функцию *ac* как распределитель памяти
`shared_ptr<`*T*`>` *sp*(`nullptr`)	Создает пустой совместно используемый указатель, используя механизм удаления, заданный по умолчанию (оператор `delete`)
`shared_ptr<`*T*`>` *sp*(`nullptr`, *del*)	Создает пустой совместно используемый указатель, используя функцию *del* в качестве оператора удаления
`shared_ptr<`*T*`>` *sp*(`nullptr`, *del*, *ac*)	Создает пустой совместно используемый указатель, используя функцию *del* в качестве оператора удаления и функцию *ac* как распределитель памяти
`shared_ptr<`*T*`>` *sp*(*sp2*)	Создает совместно используемый указатель, разделяющий владение с указателем *sp2*
`shared_ptr<`*T*`>` *sp*(`move`(*sp2*))	Создает совместно используемый указатель, владеющий указателем, которым ранее владел указатель *sp2* (указатель *sp2* после этого становится пустым)
`shared_ptr<`*T*`>` *sp*(*sp2*, *ptr*)	Псевдоним конструктора; создает совместно используемый указатель, разделяющий владение с указателем *sp2*, но ссылающийся на объект **ptr*

Продолжение табл. 5.3

Операция	Действие
`shared_ptr<T> sp(wp)`	Создает совместно используемый указатель из слабого указателя *wp*
`shared_ptr<T> sp(move(up))`	Создает совместно используемый указатель из указателя *up* класса `unique_ptr`
`shared_ptr<T> sp(move(ap))`	Создает совместно используемый указатель из указателя *ap* класса `auto_ptr`
sp.`~shared_ptr()`	Деструктор; вызывает функцию удаления, если указатель *sp* владеет объектом
sp = *sp2*	Присваивание (после него указатель *sp* разделяет владение *sp2* с указателем, отказываясь от владения объектом, которым владел ранее)
sp = `move(sp2)`	Перемещающее присваивание (указатель *sp2* передает владение указателю *sp*)
sp = `move(up)`	Присваивание указателя *up* класса `unique_ptr` (указатель *up* передает владение указателю sp)
sp = `move(ap)`	Присваивание указателя *ap* класса `auto_ptr` (указатель *ap* передает владение указателю *sp*)
sp1.`swap(sp2)`	Обмен указателей *sp1* и *sp2* и функций удаления
`swap(sp1,sp2)`	Обмен указателей *sp1* и *sp2* и функций удаления
sp.`reset()`	Передача владения и повторная инициализация совместно используемого указателя, оказавшегося пустым
sp.`reset(ptr)`	Передача владения и повторная инициализация совместно используемого указателя, владеющего объектом **ptr* с помощью функции удаления, заданной по умолчанию (выполняющей оператор `delete`)
sp.`reset(ptr,del)`	Передача владения и повторная инициализация совместно используемого указателя, владеющего объектом **ptr* с помощью функции удаления *del*
sp.`reset(ptr,del,ac)`	Передача владения и повторная инициализация совместно используемого указателя, владеющего объектом **ptr* с помощью функции удаления del и распределителя памяти *ac*
`make_shared(...)`	Создание совместно используемого указателя для нового объекта, инициализированного передаваемыми аргументами
`allocate_shared(ac,...)`	Создает совместно используемый указатель для нового объекта, инициализированного передаваемыми аргументами с помощью распределителя памяти *ac*
sp.`get()`	Возвращает хранящийся указатель (как правило, адрес объекта, которым владеет указатель, или `nullptr`, если такого объекта нет)
**sp*	Возвращает объект, которым владеет указатель (если такого объекта нет, последствия будут непредсказуемыми)
sp->...	Обеспечивает доступ к членам объекта, которым владеет указатель (если такого объекта нет, последствия будут непредсказуемыми)

Окончание табл. 5.3

Операция	Действие
sp.use_count()	Возвращает количество совместных владельцев (включая указатель *sp*) или 0, если совместно используемый указатель пуст
sp.unique()	Возвращает результат проверки, является ли указатель *sp* единственным владельцем (эквивалент проверки p.use_count()==1, но может работать быстрее)
if (*sp*)	Операция bool(); проверяет, является ли указатель *sp* пустым
sp1 == *sp2*	Вызов оператора == для хранящихся указателей (возможно, nullptr)
sp1 != *sp2*	Вызов оператора != для хранящихся указателей (возможно, nullptr)
sp1 < *sp2*	Вызов оператора < для хранящихся указателей (возможно, nullptr)
sp1 <= *sp2*	Вызов оператора <= для хранящихся указателей (возможно, nullptr)
sp1 > *sp2*	Вызов оператора > для хранящихся указателей (возможно, nullptr)
sp1 >= *sp2*	Вызов оператора >= для хранящихся указателей (возможно, nullptr)
static_pointer_cast(*sp*) static_cast<>	Семантика оператора static_cast<> для указателя *sp*
dynamic_pointer_cast(*sp*) dynamic_cast<>	Семантика оператора dynamic_cast<> для указателя *sp*
const_pointer_cast(*sp*) const_cast<>	Семантика оператора const_cast<> для указателя *sp*
get_deleter(*sp*)	Возвращает адрес функции удаления, если таковая имеется, иначе возвращает nullptr
strm << *sp*	Вызывает оператор вывода для простых указателей (эквивалентно *strm*<<*sp*.get())
sp.owner_before(*sp2*)	Устанавливает отношение строгой квазиупорядоченности с другим совместно используемым указателем
sp.owner_before(*wp*)	Устанавливает отношение строгой квазиупорядоченности с другим слабым указателем

Когда владение передается совместно используемому указателю, который уже владеет другим объектом, вызывается функция удаления объекта, которым указатель владел прежде, если указатель не был его последним владельцем. То же самое происходит, когда совместно используемый указатель получает новое значение в результате присваивания или при вызове функции reset(). Если совместно используемый указатель ранее владел объектом и был его последним владельцем, вызывается соответствующая функция удаления из этого объекта (или выполняется оператор delete). Еще раз отметим, что передаваемая функция удаления не должна генерировать исключений.

Совместно используемые указатели могут использовать разные типы объектов, при условии, что разрешено неявное преобразование указателя. По этой причине конструкторы, оператор присваивания и функция-член `reset()` являются шаблонными, а операторы сравнения имеют разные шаблонные типы.

Все операторы сравнения сравнивают простые указатели, скрытые в совместно используемых указателях (иначе говоря, они применяют один и тот же оператор к значениям, возвращенным функцией `get()`). Все они имеют перегруженные варианты, в которых аргументом является значение `nullptr`. Таким образом, можно проверить корректность указателя и сравнить простой указатель со значением `nullptr`.

Конструктор, получающий аргумент типа `weak_ptr`, генерирует исключение `bad_weak_ptr` (см. раздел 5.2.2), если слабый указатель пуст (функция `expired()` возвращает `true`).

Функция `get_deleter()` возвращает указатель на функцию, предназначенную для реализации оператора удаления (если таковая имеется), иначе возвращает `nullptr`. Указатель остается корректным, пока совместно используемый указатель владеет функцией удаления. Однако, для того чтобы получить функцию удаления, необходимо передать ее тип в качестве шаблонного аргумента. Например:

```
auto del = [] (int* p) {
              delete p;
           };
std::shared_ptr<int> p(new int, del);
decltype(del)* pd = std::get_deleter<decltype(del)>(p);
```

Отметим, что совместно используемые указатели не предоставляют оператор `release()` для отказа от владения и возврата управления объектом вызывающей программе, потому что другие совместно используемые указатели все еще могут владеть объектом.

Более сложные операции с указателями класса shared_ptr

Некоторые операции над совместно используемыми указателями не так очевидны. Обоснование и описание большинства из них изложено в документе [N2351:SharedPtr].

Конструктор, получающий другой совместно используемый указатель и дополнительный простой указатель, называется *конструктором псевдонима* (aliasing constructor), что позволяет отразить тот факт, что одним объектом владеет другой объект. Например:

```
struct X
{
   int a;
};
shared_ptr<X> px(new X);
shared_ptr<int> pi(px, &px->a);
```

Объект типа `X` "владеет" своим членом `a`, поэтому, для того, чтобы создать совместно используемый указатель на `a`, необходимо сохранить в целости окружающий его объект, присоединив к нему счетчик ссылок с помощью конструктора псевдонима. Существуют и другие, более сложные примеры, например, ссылка на элемент контейнера или символ совместно используемой библиотеки[11].

[11] Благодарю Петера Димова (Peter Dimov) за это замечание.

Отметим, что вследствие вышесказанного программист должен гарантировать, что время жизни обоих объектов совпадает. В противном случае могут возникнуть висячие указатели или утечка ресурсов. Например:

```
shared_ptr<X> sp1(new X);
shared_ptr<X> sp2(sp1,new X); // ОШИБКА: оператор delete для этого
                              // объекта X никогда не выполняется

sp1.reset();                  // удаляет первый объект X;
                              // делает указатель sp1 пустым
shared_ptr<X> sp3(sp1,new X); // use_count()==0, но get()!=nullptr
```

Функции `make_shared()` и `allocate_shared()` предназначены для оптимизации процесса создания совместно используемого указателя и связанного с ним управляющего блока (например, блока, поддерживающего счетчик ссылок). Отметим, что инструкция

```
shared_ptr<X>(new X(...))
```

выполняет выделение памяти дважды: один раз для объекта `X`, а второй — для управляющего блока, используемого, например, совместно используемым указателем для управления счетчиком. Операция

```
make_shared<X>(...)
```

работает значительно быстрее, выполняя только одно выделение памяти, и в то же время он оказывается безопаснее в ситуациях, когда выделение памяти для объекта `X` выполнено успешно, а выделение памяти для управляющего блока завершается неудачно. Функция `allocate_shared()` позволяет передавать в качестве первого аргумента свой собственный распределитель памяти.

Операторы приведения типа позволяют приводить указатель к разным типам. Их семантика совпадает с семантикой соответствующих операторов, а результатом является совместно используемый указатель другого типа. Отметим, что использовать для этого обычные операторы приведения невозможно, поскольку они приводят к непредсказуемым результатам.

```
shared_ptr<void> sp(new int); // совместно используемый указатель хранит внутри
                              // себя обычный указатель void*
...
shared_ptr<int>(static_cast<int*>(sp.get())) // ОШИБКА: неопределенное поведение
static_pointer_cast<int>(sp)                 // OK
```

Подробное описание класса `weak_ptr`

Как указано в разделе 5.2.2, класс `weak_ptr` является вспомогательным классом, позволяющим классу `shared_ptr` совместно управлять объектом, не владея им. Его функция-член `use_count()` возвращает количество указателей класса `shared_ptr`, владеющих объектом, для которого указатели класса `weak_ptr`, управляющие объектом, владельцами не являются. Кроме того, указатель класса `weak_ptr` может быть пустым, например, если он не инициализирован указателем класса `shared_ptr` или если последний владелец соответствующего объекта был удален. Класс `weak_ptr<>` имеет шаблонный параметр — тип объекта, на который ссылается исходный указатель.

```
namespace std {
   template <typename T>
   class weak_ptr
   {
      public:
         typedef T element_type;
         ...
      };
}
```

Все операции над слабыми указателями приведены в табл. 5.4.

Таблица 5.4. Операции над указателями класса `weak_ptr`

Операция	Действие
`weak_ptr<`*T*`>` *wp*	Конструктор по умолчанию; создает пустой слабый указатель
`weak_ptr<`*T*`>` *wp*`(`*sp*`)`	Создает слабый указатель, разделяющий владение с указателем, принадлежащим объекту *sp*
`weak_ptr<`*T*`>` *wp*`(`*wp2*`)`	Создает слабый указатель, разделяющий владение с указателем, принадлежащим объекту *wp2*
wp`.~weak_ptr()`	Деструктор; уничтожает слабый указатель, не влияя на объект, которым он владел
wp = *wp2*	Присваивание (после него указатель *wp* разделяет владение с указателем *wp2*, отказываясь от владения объектом, которым владел ранее)
wp = *sp*	Присваивает совместно используемый указатель *sp* (после этого указатель *wp* разделяет владение с указателем *sp*, отказываясь от владения объектом, которым владел ранее)
wp`.swap(`*wp2*`)`	Обмен указателей *wp* и *wp2*
`swap(`*wp1*`,`*wp2*`)`	Обмен указателей *wp1* и *wp2*
wp`.reset()`	Отказ от владения объектом, которым владел ранее (если таковой имелся), и повторная инициализация пустого слабого указателя
wp`.use_count()`	Возвращает количество владельцев совместно используемых указателей (указателей `shared_ptr`, владеющих объектом) или `0`, если слабый указатель пуст
wp`.expired()`	Возвращает результат проверки, является ли указатель *wp* пустым (эквивалент выражения *wp*`.use_count()==0`, иногда работающий быстрее)
wp`.lock()`	Возвращает совместно используемый указатель, владеющий указателем совместно со слабым указателем (или пустой совместно используемый указатель, если объекта владения не существует)
wp`.owner_before(`*wp2*`)`	Выполняет строгое квазиупорядочение с другим слабым указателем
wp`.owner_before(`*sp*`)`	Выполняет строгое квазиупорядочение с совместно используемым указателем

Конструктор, заданный по умолчанию, создает пустой слабый указатель. Это значит, что функция `expired()` возвращает значение `true`. Поскольку функция `lock()` возвращает совместно используемый указатель, счетчик использования объекта увеличивает значение на единицу на время существования совместно используемого указателя. Это единственный способ для работы с объектом, которым управляет слабый указатель.

Потокобезопасный интерфейс совместно используемого указателя

В целом совместно используемые указатели не являются потокобезопасными. По этой причине, чтобы избежать непредсказуемых последствий гонки за данными (см. раздел 18.4.1), в случае, когда совместно используемые указатели ссылаются на один и тот же объект в нескольких потоках, необходимо использовать специальные средства, такие как мьютексы и блокировки. Однако чтение счетчика использований в момент, когда его модифицирует другой поток, еще не создает гонки за данными, хотя значение счетчика может оказаться устаревшим. Фактически один поток может проверять счетчик использования, в то время как другой поток манипулирует им. Подробности описаны в главе 18.

В стандартной библиотеке C++ предусмотрены перегруженные версии для совместно используемых указателей, позволяющие работать с ними параллельно и соответствующие атомарному интерфейсу обычных указателей в стиле языка C (см. раздел 18.7.3). Отметим, что в этом случае подразумевается параллельный доступ к *указателям*, а не к значениям, на которые они ссылаются.

Рассмотрим пример[12]:

```
std::shared_ptr<X> global; // начальное значение равно nullptr
void foo()
{
   std::shared_ptr<X> local(new X);
   ...
   std::atomic_store(&global,local);
}
```

Высокоуровневый интерфейс описан в табл. 5.5. В библиотеке также предусмотрен низкоуровневый интерфейс (см. раздел 18.7.4).

Таблица 5.5. Высокоуровневый интерфейс атомарных операций `shared_ptr`

Операция	Действие
`atomic_is_lock_free(&sp)`	Возвращает значение `true`, если атомарный интерфейс к указателю *sp* не блокирован
`atomic_load(&sp)`	Возвращает указатель *sp*
`atomic_store(&sp,sp2)`	Присваивает указатель *sp2* указателю *sp*
`atomic_exchange(&sp,sp2)`	Обменивает значение указателей *sp* и *sp2*

5.2.5. Класс `unique_ptr`

Тип `unique_ptr`, предусмотренный в стандартной библиотеке C++ в соответствии со стандартом C++11, представляет собой разновидность интеллектуального указателя,

[12] Благодарю за этот пример Энтони Уильямса (Anthony Williams).

позволяющего избежать утечки ресурсов при генерировании исключений. Этот интеллектуальный указатель реализует концепцию *исключительного владения* (exclusive ownership), т.е. он гарантирует, что объект и связанные с ним ресурсы в каждый момент принадлежат только одному указателю. Если владелец объекта был удален, стал пустым или стал владеть другим объектом, ранее принадлежавший ему объект также удаляется, а все связанные с ним ресурсы освобождаются.

Класс `unique_ptr` является аналогом класса `auto_ptr`, включенного в стандарт C++98, но в настоящее время считающийся устаревшим (см. раздел 5.2.7). Класс `unique_ptr` предоставляет простой и ясный интерфейс, менее подверженный ошибкам, чем интерфейс класса `auto_ptr`.

Предназначение класса `unique_ptr`

Функции часто работают так, как описано ниже[13]:

1. Захватывают некоторые ресурсы.
2. Выполняют определенные операции.
3. Освобождают захваченные ресурсы.

При работе с локальными объектами ресурсы, выделенные при входе в функцию, автоматически освобождаются при выходе из нее, потому что при этом вызываются деструкторы локальных объектов. Однако, если ресурсы были получены явно и не связаны ни с каким объектом, они также должны быть явно освобождены. При использовании указателей ресурсы обычно управляются явно.

Типичным примером такого использования указателей является выполнение операторов `new` и `delete` для создания и удаления объекта.

```
void f()
{
   ClassA* ptr = new ClassA; // явно создает объект
   ...                       // выполняет некоторые операции
   delete ptr;               // очистка (явно уничтожает объект)
}
```

Эта функция является источником неприятностей. Одна из очевидных проблем заключается в том, что программист может забыть удалить объект, особенно если в функции выполняется оператор `return`. Менее очевидная опасность кроется при генерировании исключения. Такое исключение может привести к немедленному выходу из функции, без выполнения оператора `delete`, расположенной в конце функции. В результате возникнет утечка памяти или, в более общем случае, утечка ресурсов.

Для предотвращения утечки ресурсов обычно необходим перехват всех исключений в функции. Например:

```
void f()
{
```

[13] Это описание, изначально написанное для класса `auto_ptr`, позаимствовано с разрешения автора, Скотта Мейерса (Scott Meyers), из книги *More Effective C++*. Общий принцип изначально был сформулирован Бьярне Страуструпом (Bjarne Stroustrup) в виде идиомы "захват ресурса является инициализацией" в книгах *The C++ Programming Language, 2nd edition* и *The Design and Evolution of C++*.

```
   ClassA* ptr = new ClassA; // явно создает объект
   try {
      ...                    // выполняет некоторые операции
   }
   catch (...) {             // для любого исключения
      delete ptr;            // - очистка
      throw;                 // - повторное генерирование исключения
   }

   delete ptr;               // очистка в конце
}
```

Код, предназначенный для правильного удаления данного объекта при возникновении исключения, стал более сложным и громоздким. Если мы будем удалять и второй объект таким же способом или напишем несколько разделов `catch`, проблема лишь усугубится. Это плохой стиль программирования, которого следует избегать, потому что он сложен и уязвим для ошибок.

В данном случае может помочь интеллектуальный указатель. При уничтожении интеллектуального указателя он может освобождать данные, на которые ссылался. Более того, поскольку он является локальной переменной, указатель будет удален автоматически при выходе из функции, независимо от того, был ли выход нормальным или возникло исключение. Такой интеллектуальный указатель описывается классом `unique_ptr`.

Объект класса `unique_ptr` — это указатель, служащий единственным *владельцем* объекта, на который он ссылается. В результате при уничтожении указателя класса `unique_ptr` объект удаляется автоматически. Класс `unique_ptr` имеет единственное требование — чтобы объект мог иметь только одного владельца.

Перепишем предыдущий пример с помощью класса `unique_ptr`.

```
// заголовочный файл для класса unique_ptr
#include <memory>

void f()
{
   // создает и инициализирует объект класса unique_ptr
   std::unique_ptr<ClassA> ptr(new ClassA);
   ...                          // выполняет некоторые операции
}
```

Вот и все. Оператор `delete` и раздел `catch` больше не нужны.

Использование класса `unique_ptr`

Интерфейс класса `unique_ptr` мало отличается от обычного указателя, т.е. оператор * разыменовывает объект, на который ссылается, а оператор -> обеспечивает доступ к члену объекта, являющегося экземпляром класса или структуры.

```
// создам и инициализируем (устанавливаем) указатель на строку:
std::unique_ptr<std::string> up(new std::string("nico"));

(*up)[0] = 'N';                  // заменяем первый символ
up->append("lai");               // добавляем несколько символов
std::cout << *up << std::endl;   // выводим всю строку
```

Однако для класса unique_ptr не определена арифметика указателей, например, оператор ++ (это можно считать преимуществом, потому что арифметика указателей является источником неприятностей).

Отметим, что класс unique_ptr<> не позволяет инициализировать объект обычным указателем, используя синтаксис присваивания. Таким образом, объект класса unique_ptr следует инициализировать явно, используя его значение.

```
std::unique_ptr<int> up = new int; // ОШИБКА
std::unique_ptr<int> up(new int);  // ОК
```

Указатель класса unique_ptr не обязан владеть объектом, поэтому он может быть *пустым*[14]. Это происходит, например, когда он инициализируется конструктором, заданным по умолчанию.

```
std::unique_ptr<std::string> up;
```

Указателю unique_ptr можно присвоить значение nullptr или вызвать для него функцию reset().

```
up = nullptr;
up.reset();
```

Кроме того, можно вызвать функцию release(), возвращающую объект, которым владеет указатель unique_ptr, и при этом выполняющую отказ от владения, так что за объект теперь отвечает вызывающий код.

```
std::unique_ptr<std::string> up(new std::string("nico"));
...
std::string* sp = up.release(); // потеря владения
```

Для того чтобы проверить, владеет ли объектом уникальный указатель, можно выполнить оператор bool():

```
if (up) { // если указатель up не пуст
    std::cout << *up << std::endl;
}
```

Вместо этого можно сравнить уникальный указатель со значением nullptr или запросить обычный указатель, находящийся внутри указателя unique_ptr и имеющий значение nullptr, если указатель unique_ptr не владеет ни одним объектом.

```
if (up != nullptr)       // если указатель up не пуст
if (up.get() != nullptr) // если указатель up не пуст
```

Передача владения указателем unique_ptr

Класс unique_ptr обеспечивает семантику исключительного владения. Однако программист должен самостоятельно гарантировать, что два уникальных указателя не инициализированы одним и тем же указателем.

[14] Несмотря на то что в стандартной библиотеке языка C++ термин *пустой* определен только для разделяемых указателей, я не вижу причин использовать его в общем смысле.

```
std::string* sp = new std::string("hello");
std::unique_ptr<std::string> up1(sp);
std::unique_ptr<std::string> up2(sp); // ОШИБКА: up1 и up2 владеют
                                      // одними и теми же данными
```

К сожалению, эта ошибка возникает на этапе выполнения, а не компиляции программы, поэтому программист должен заранее ее избежать.

Все это приводит к вопросу: как действуют копирующий конструктор и оператор присваивания в классе `unique_ptr`? Ответ простой: нельзя копировать или присваивать уникальный указатель, используя обычную семантику копирования. Однако можно использовать семантику перемещения, предусмотренную в стандарте C++11 (см. раздел 3.1.5). В этом случае конструктор или оператор присваивания *передает* владение другому уникальному указателю[15].

Рассмотрим следующий пример использования копирующего конструктора:

```
// инициализация указателя unique_ptr новым объектом
std::unique_ptr<ClassA> up1(new ClassA);
// копирование указателя unique_ptr
std::unique_ptr<ClassA> up2(up1);             // ОШИБКА: невозможно
// передача владения указателю unique_ptr
std::unique_ptr<ClassA> up3(std::move(up1)); // OK
```

После выполнения первого оператора указатель `up1` владеет объектом, созданным с помощью оператора `new`. Второй оператор, пытающийся вызвать копирующий конструктор, вызывает ошибку во время компиляции, потому что указатель `up2` не может стать владельцем этого объекта. В каждый момент времени может существовать только один владелец этого объекта. Однако третий оператор передает владение от указателя `up1` указателю `up3`. После этого указатель `up3` вступает во владение объектом, созданным оператора `new`, а указатель `up1` больше не владеет этим объектом. Объект, созданный оператором `new ClassA`, уничтожается только один раз — когда уничтожается указатель `up3`.

Оператор присваивания работает аналогично:

```
// инициализация указателя класса unique_ptr новым объектом
std::unique_ptr<ClassA> up1(new ClassA);
std::unique_ptr<ClassA> up2; // создаем новый указатель класса unique_ptr
up2 = up1;                   // ОШИБКА: невозможно
up2 = std::move(up1);        // присваиваем указатель класса unique_ptr
                             // - передаем владение от указателя up1
                             // указателю up2
```

Здесь перемещающее присваивание передает владение от указателя `up1` указателю `up2`. В результате указатель `up2` вступает во владение объектом, которым ранее владел указатель `up1`.

Если указатель `up2` владел объектом до присваивания, к этому объекту применяется оператор `delete`.

```
// инициализация указателя класса unique_ptr новым объектом
std::unique_ptr<ClassA> up1(new ClassA);
// инициализация другого указателя класса unique_ptr новым объектом
```

[15] В этом заключается его главное отличие от класса `auto_ptr`, в котором передача владения осуществляется с помощью обычной семантики копирования, что приводит к неприятностям и путанице.

```
std::unique_ptr<ClassA> up2(new ClassA);
up2 = std::move(up1); // перемещающее присваивание указателя класса unique_ptr
                      // - удаление объекта, которым владел указатель up2
                      // - передача владения от указателя up1 указателю up2
```

Указатель `unique_ptr`, потерявший владение объектом и не вступивший во владение новым объектом, не ссылается ни на один объект.

Для присвоения нового значения указателю `unique_ptr` это новое значение также должно иметь тип `unique_ptr`. Указателю `unique_ptr` нельзя присваивать обычный указатель.

```
std::unique_ptr<ClassA> ptr;                  // создание указателя unique_ptr
ptr = new ClassA;                             // ОШИБКА
ptr = std::unique_ptr<ClassA>(new ClassA); // OK, удаляем объект, которым
                                              // владеет указатель ptr,
                                              // и вступаем во владение новым
```

Присваивание значения `nullptr` тоже возможно. Оно имеет тот же эффект, что и вызов функции `reset()`.

```
up = nullptr; // удаляет соответствующий объект, если он существует
```

Источник и сток

Передача владения подразумевает специальное использование указателей класса `unique_ptr`; иначе говоря, функции могут использовать их для передачи владения другим функциям. Это можно сделать двумя способами.

1. Функция может играть роль *стока данных* (sink of data). Это происходит, когда указатель класса `unique_ptr` передается как аргумент в функцию с помощью rvalue-ссылки, созданной функцией `std::move()`. В этом случае параметр вызываемой функции получает владение указателя `unique_ptr`. Таким образом, если функция не передаст его снова, объект будет удален при выходе из нее.

   ```
   void sink(std::unique_ptr<ClassA> up) // функция sink() получает владение
   {
      ...

   }
   std::unique_ptr<ClassA> up(new ClassA);
      ...
   sink(std::move(up));                  // потеря владения
     ...
   ```

2. Функция может играть роль *источника* данных (source of data). Если возвращается указатель `unique_ptr`, владение возвращаемым значением передается в вызывающий контекст. Этот способ демонстрирует следующий пример:[16]

   ```
   std::unique_ptr<ClassA> source()
   {
   ```

[16] Если вы собираетесь объявить тип возвращаемого значения как rvalue-ссылку, не делайте этого, так как это может привести к висячему указателю (см. раздел 3.1.5).

```
    std::unique_ptr<ClassA> ptr(new ClassA); // указатель ptr
                                             // владеет новым объектом
    ...
    return ptr; // передача владения в вызывающую функцию
}

void g()
{
    std::unique_ptr<ClassA> p;

    for (int i=0; i<10; ++i) {
       p = source(); // указатель p получает владение
                     // возвращенным объектом
                     // (объект, ранее возвращенный
                     // функцией source() удаляется)
      ...
    }
} // удаляется последний объект, которым владел указатель p
```

Каждый раз при вызове функции `source()` она создает объект с помощью оператора `new` и возвращает этот объект вместе с его владением в вызывающую функцию. Присвоение возвращенного значения указателю `p` передает владение указателю `p`. При втором и последующих выполнениях цикла присваивание указателю `p` удаляет объект, которым до этого владел указатель `p`. При выходе из функции `g()` и соответствующем удалении указателя `p` последний объект, которым владел указатель `p`, удаляется. В любом случае утечка ресурсов невозможна. Даже если возникнет исключение, любой указатель `unique_ptr`, владеющий данными, гарантирует их удаление.

Причина, по которой в операторе `return` в функции `source()` не обязательно использовать функцию `std::move()`, заключается в том, что по правилам стандарта C++11 компилятор попытается выполнить перенос автоматически (см. раздел 3.1.5).

Указатели типа `unique_ptrs` как члены других классов

Используя указатели типа `unique_ptr` в классе, также можно избежать утечки ресурсов. Если вместо обычного указателя использовать указатель типа `unique_ptr`, деструктор больше не понадобится, потому что объект будет уничтожен в ходе удаления этого указателя. Кроме того, класс `unique_ptr` позволяет избежать утечки ресурсов, вызванной исключениями во время инициализации объекта. Отметим, что деструкторы вызываются только после полного завершения процесса создания объекта. Следовательно, если в конструкторе возникнет исключение, деструктор будет вызван только тогда, когда объект, являющийся членом класса, будет полностью создан. Это может привести к утечке ресурсов для класса с многочисленными простыми указателями, если в процессе создания его объекта первое выполнение оператора `new` окажется успешным, а второе нет. Рассмотрим пример:

```
class ClassB {
  private:
    ClassA* ptr1;     // указатели - члены класса
    ClassA* ptr2;
  public:
    // конструктор, инициализирующий указатели
    // - вызывает утечку ресурсов, если второй оператор new
```

```
      //   генерирует исключение
   ClassB (int val1, int val2)
    : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {
   }

   // копирующий конструктор
   // - вызывает утечку ресурсов, если второй оператор new
   //   генерирует исключение
   ClassB (const ClassB& x)
    : ptr1(new ClassA(*x.ptr1)), ptr2(new ClassA(*x.ptr2)) {
   }

   // оператор присваивания
   const ClassB& operator= (const ClassB& x) {
      *ptr1 = *x.ptr1;
      *ptr2 = *x.ptr2;
      return *this;
   }

   ~ClassB () {
      delete ptr1;
      delete ptr2;
   }
   ...
};
```

Для того чтобы предотвратить возможную утечку ресурсов, можно использовать указатели типа `unique_ptr`.

```
class ClassB {
   private:
      std::unique_ptr<ClassA> ptr1; // указатели типа unique_ptr -
                                    // члены класса
      std::unique_ptr<ClassA> ptr2;
   public:
      // конструктор, инициализирующий указатели типа,
      // - утечка ресурсов невозможна
   ClassB (int val1, int val2)
    : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {
   }

   // копирующий конструктор
   // - утечка ресурсов невозможна
   ClassB (const ClassB& x)
    : ptr1(new ClassA(*x.ptr1)), ptr2(new ClassA(*x.ptr2)) {
   }

   // оператор присваивания
   const ClassB& operator= (const ClassB& x) {
   *ptr1 = *x.ptr1;
   *ptr2 = *x.ptr2;
   return *this;
   }
```

```
    // деструктор не нужен
    // (деструктор по умолчанию разрешает указателям
    // ptr1 и ptr2 удалить свои объекты)
    ...
};
```

Отметим, что теперь можно опустить деструктор, потому что его работу выполнят указатели типа `unique_ptr`. Кроме того, необходимо реализовать копирующий конструктор и оператор присваивания. По умолчанию оба они пытаются копировать или присваивать члены класса, что в данном случае невозможно. Если их не предусмотреть, то конструктор класса `ClassB` также будет реализовывать только семантику перемещения.

Работа с массивами

По умолчанию указатели типа `unique_ptr` применяют оператор `delete` для объекта, которым владеют, если они потеряли владение (т.е. если они были удалены, им присвоены новые объекты или они стали пустыми). К сожалению, из-за правил, унаследованных от языка C, язык C++ неспособен различать типы указателей на отдельный объект и на массив объектов. Однако по языковым правилам, предусмотренным для массивов, их следует удалять с помощью оператора `delete[]`, а не `delete`. Следовательно, следующий код возможен, но некорректен:

```
std::unique_ptr<std::string> up(new std::string[10]); // ОШИБКА
                                                      // времени выполнения
```

Можно предположить, что по аналогии с классом `shared_ptr` (см. раздел 5.2.1) мы должны определить собственный оператор удаления для массивов. Однако это не обязательно.

К счастью, стандартная библиотека языка C++ содержит частичную специализацию класса `unique_ptr` для массивов, которая выполняет оператор `delete[]` для объекта, на который ссылается указатель, когда последний теряет владение этим объектом. Итак, следует лишь объявить такой указатель:

```
std::unique_ptr<std::string[]> up(new std::string[10]); // OK
```

Отметим, однако, что эта частичная специализация имеет немного другой интерфейс. Вместо операторов `*` и `->` доступ к отдельному объекту в массиве, на который ссылается указатель, предоставляет оператор `[]`.

```
std::unique_ptr<std::string[]> up(new std::string[10]); // OK
...
std::cout << *up << std::endl;   // ОШИБКА: оператор * для массивов
                                 // не определен
std::cout << up[0] << std::endl; // OK
```

Как обычно, ответственность за корректность индекса лежит на программисте. Использование некорректного индекса приводит к непредсказуемым последствиям.

Отметим также, что этот класс не позволяет инициализировать объект массивом производного типа. Это отражает тот факт, что полиморфизм не распространяется на обычные массивы.

Класс default_delete<>

Остановимся на объявлении класса unique_ptr. Концептуально этот класс объявлен следующим образом[17]:

```
namespace std {
   // первичный шаблон:
   template <typename T, typename D = default_delete<T>>
   class unique_ptr
   {
      public:
      ...
      T& operator*() const;
      T* operator->() const noexcept;
      ...
   };

   // частичная специализация для массивов:
   template<typename T, typename D>
   class unique_ptr<T[], D>
   {
      public:
      ...
      T& operator[](size_t i) const;
      ...
   }
}
```

Как видим, существует специализированная версия класса unique_ptr для работы с массивами. Эта версия содержит оператор [], который заменяет операторы * и -> при работе с массивами, а не отдельными объектами. Однако обе эти версии используют в качестве оператора удаления функцию std::default_delete<>, которая сама специализирована для применения к массивам оператора delete[], а не delete.

```
namespace std {
   // первичный шаблон:
   template <typename T> class default_delete {
      public:
         void operator()(T* p) const; // выполняет оператор delete p
         ...
      };

  // частичная специализация для массивов:
  template <typename T> class default_delete<T[]> {
      public:
         void operator()(T* p) const; // выполняет оператор delete[] p
      ...
   };
}
```

[17] В стандартной библиотеке языка C++ класс unique_ptr на самом деле намного сложнее, потому что в нем используется механизм шаблонов для указания типа значения, возвращаемого операциями * и ->.

Отметим, что шаблонные аргументы, заданные по умолчанию, автоматически применяются и для частичных специализаций.

Операторы удаления для других связанных ресурсов

Когда объект, на который вы ссылаетесь, требует выполнить что-нибудь другое, а не операторы `delete` или `delete[]`, необходимо задать свой собственный оператор удаления. Тем не менее отметим, что в данном случае применяется другой подход к определению оператора удаления по сравнению с классом `shared_ptr`. Тип оператора удаления следует задать в качестве второго шаблонного аргумента. Этот тип может быть ссылкой на функцию, указателем на функцию или функциональным объектом (см. раздел 6.10). Если используется функциональный объект, то его "оператор вызова функции" `()` должен быть объявлен как получающий указатель на объект.

Например, следующий код выводит на экран дополнительное сообщение, прежде чем удалить объект с помощью оператора `delete`:

```
class ClassADeleter
{
   public:
      void operator () (ClassA* p) {
         std::cout << "вызов delete для объекта класса ClassA" << std::endl;
         delete p;
      }
};
   ...
std::unique_ptr<ClassA,ClassADeleter> up(new ClassA());
```

Для задания функции или лямбда-функции необходимо объявить тип оператора удаления как `void(*)(T*)` или `std::function<void(T*)>` или же использовать ключевое слово `decltype` (см. раздел 3.1.11). Например, чтобы использовать собственный оператор удаления для массива чисел типа `int`, заданный с помощью лямбда-функции, следует написать

```
std::unique_ptr<int,void(*)(int*)> up(new int[10],
                                      [](int* p) {
                                         ...
                                         delete[] p;
                                      });
```

или

```
std::unique_ptr<int,std::function<void(int*)>> up(new int[10],
                                                   [](int* p) {
                                                   ...
                                                   delete[] p;
                                                   });
```

или

```
auto l = [](int* p) {
            ...
            delete[] p;
```

```
            };
std::unique_ptr<int,decltype(l)>> up(new int[10], l);
```

Для того чтобы избежать задания *типа* оператора удаления при передаче указателя на функцию или лямбда-функции, можно также использовать шаблонный псевдоним — языковое средство, появившееся в стандарте C++11 (см. раздел 3.1.9).

```
template <typename T>
using uniquePtr = std::unique_ptr<T,void(*)(T*)>; // шаблонный псевдоним
...

uniquePtr<int> up(new int[10], [](int* p) { // используется здесь
                                   ...
                                   delete[] p;
                                   });
```

Таким образом, мы получаем интерфейс, более или менее похожий на интерфейс оператора удаления в классе `shared_ptr`.

Ниже приведен полный пример, демонстрирующий использование собственного оператора удаления.

```
// util/uniqueptr1.cpp
#include <iostream>
#include <string>
#include <memory>    // для класса unique_ptr
#include <dirent.h>  // для функции opendir(), ...
#include <cstring>   // для функции strerror()
#include <cerrno>    // для потока errno
using namespace std;

class DirCloser
{
   public:
      void operator () (DIR* dp) {
          if (closedir(dp) != 0) {
              std::cerr << "OOPS: closedir() failed" << std::endl;
          }
   }
};

int main()
{
   // открываем текущий каталог:
   unique_ptr<DIR,DirCloser> pDir(opendir("."));

   // обрабатываем каждую запись в каталоге:
   struct dirent *dp;
   while ((dp = readdir(pDir.get())) != nullptr) {
        string filename(dp->d_name);
        cout << "process " << filename << endl;
        ...
   }
}
```

В функции `main()` мы обрабатываем записи текущего каталога, используя стандартный POSIX-интерфейс функций `opendir()`, `readdir()` и `closedir()`. Для того чтобы открытый каталог всегда закрывался с помощью функции `closedir()`, мы определили объект класса `unique_ptr`, вызывающий функцию-объект `DirCloser` при удалении дескриптора, ссылающегося на открытый каталог. Как и в классе совместно используемых указателей, операторы удаления для уникальных указателей не должны генерировать исключений. По этой причине мы выводим сообщение об ошибке.

Другое преимущество класса `unique_ptr` заключается в невозможности копирования его объектов. Отметим, что функция `readdir()` имеет состояние, поэтому целесообразно гарантировать, чтобы при работе дескриптора с каталогом копия дескриптора не модифицировала ее состояние.

Если вы не хотите обрабатывать значение, возвращаемое функцией `closedir()`, то можете передать эту функцию с помощью указателя, задав оператор удаления с помощью указателя на функцию. Но будьте осторожны: часто рекомендуемое объявление

```
unique_ptr<DIR,int(*)(DIR*)> pDir(opendir("."),
                                  closedir); // может не работать
```

не гарантирует переносимости, поскольку функция `closedir` имеет связывание `extern "C"` и в коде на языке C++ не всегда преобразовывается в `int(*)(DIR*)`. Для обеспечения переносимости кода необходимо промежуточное определение типа[18].

```
extern "C" typedef int(*DIRDeleter)(DIR*);
unique_ptr<DIR, DIRDeleter> pDir(opendir("."),
                                 closedir); // OK
```

Отметим, что функция `closedir()` возвращает значение типа `int`, поэтому в качестве типа оператора удаления необходимо указать `int(*)(DIR*)`. Кроме того, вызов с помощью указателя на функцию является косвенным, а значит, усложняет оптимизацию.

Другой пример использования собственного оператора удаления указателя типа `unique_ptr` для восстановления перенаправляемого буфера вывода описан в разделе 15.12.3.

5.2.6. Подробное описание класса `unique_ptr`

Как указано в разделе 5.2.5, класс `unique_ptr` реализует концепцию интеллектуального указателя с семантикой *исключительного владения*. Как только интеллектуальный указатель получил исключительное управление, вы не можете (случайно) создать ситуацию, в которой ассоциированным объектом будут владеть сразу несколько указателей. Основная цель этого класса — гарантировать, что в конце существования указателя связанный с ним объект будет удален (или его ресурсы будут освобождены). В частности, это позволяет обеспечить безопасность исключений. В противоположность совместно используемым указателям, уникальные указатели требуют минимальных дополнительных затрат памяти и времени.

Шаблонным параметром класса `unique_ptr<>` является тип объекта, на который ссылается указатель и его оператор удаления.

```
namespace std {
   template <typename T, typename D = default_delete<T>>
```

[18] Благодарю за это замечание Дэниеля Крюглера.

```
   class unique_ptr
   {
      public:
         typedef ... pointer; // возможно, D::pointer
         typedef T element_type;
         typedef D deleter_type;
         ...
   };
}
```

В библиотеке предусмотрена частичная специализация для массивов (отметим, что по правилам языка она содержит тот же самый оператор удаления, предусмотренный по умолчанию: `default_delete<T[]>`).

```
namespace std {
   template <typename T, typename D>
   class unique_ptr<T[], D>
   {
      public:
      typedef ... pointer;     // возможно, D::pointer
      typedef T element_type;
      typedef D deleter_type;
      ...
   };
}
```

Тип элемента `T` может быть `void`. В таком случае уникальный указатель владеет объектом неопределенного типа, подобно указателю типа `void*`. Отметим, что в классе определен тип `pointer`, который не обязательно определен как тип `T*`. Если оператор удаления `D` имеет определение типа `pointer`, будет использоваться именно этот тип. В таком случае шаблонный параметр `T` играет лишь роль дескриптора типа, потому что ни один член класса `unique_ptr<>` не зависит от типа `T`; все они зависят от типа `pointer`. Преимущество заключается в том, что класс `unique_ptr` в этом случае может хранить другие интеллектуальные указатели.

Если указатель класса `unique_ptr` является пустым, он не владеет ни одним объектом, поэтому функция-член `get()` возвращает значение `nullptr`.

Все операции над уникальными указателями приведены в табл. 5.6.

Таблица 5.6. Операции над объектами класса `unique_ptr`

Операция	Действие
`unique_ptr<...>` *up*	Конструктор по умолчанию; создает пустой уникальный указатель, используя в качестве оператор удаления экземпляр типа, заданного по умолчанию или переданного как шаблонный параметр
`unique_ptr<`*T*`>` *up*`(nullptr)`	Создает пустой уникальный указатель, используя в качестве оператор удаления экземпляр типа, заданного по умолчанию или переданного как шаблонный параметр
`unique_ptr<...>` *up*`(`*ptr*`)`	Создает уникальный указатель, владеющий объектом **ptr*, используя в качестве оператор удаления экземпляр типа, заданного по умолчанию или переданного как шаблонный параметр

Продолжение табл. 5.6

Операция	Действие
`unique_ptr<...>` *up*`(`*ptr*`,` *del*`)`	Создает уникальный указатель, владеющий объектом **ptr*, используя в качестве функтора удаления аргумент *del*
`unique_ptr<`*T*`>` *up*`(move(`*up2*`))`	Создает уникальный указатель, владеющий указателем, которым ранее владел указатель *up2* (после выполнения указатель *up2* становится пустым)
`unique_ptr<T>` *up*`(move(`*ap*`))`	Создает уникальный указатель, владеющий указателем, которым ранее владел указатель `auto_ptr` *ap* (после выполнения указатель *ap* становится пустым)
up`.~unique_ptr()`	Деструктор; выполняет оператор удаления объекта, являющегося предметом владения
up `= move(`*up2*`)`	Перемещающее присваивание (указатель *up2* передает владение указателю *up*)
up `= nullptr`	Выполняет оператор удаления объекта, являющегося предметом владения, и делает указатель *up* пустым (эквивалентно *up*`.reset()`)
up1`.swap(`*up2*`)`	Обменивает указатели и операторы удаления указателей *up1* и *up2*
`swap(`*up1*`,` *up2*`)`	Обменивает указатели и операторы удаления *up1* и *up2*
up`.reset()`	Выполняет оператор удаления объекта, являющегося предметом владения, и делает указатель *up* пустым (эквивалентно *up*`=nullptr`)
up`.reset(`*ptr*`)`	Выполняет оператор удаления объекта, являющегося предметом владения, и выполняет повторную инициализацию совместно с объектом, который используется указателем **ptr*
up`.release()`	Возвращает владение вызывающему коду (возвращает объект, являющийся предметом владения, не выполняя оператор удаления)
up`.get()`	Возвращает хранимый указатель (адрес объекта, являющегося предметом владения, или значение `nullptr`, если такого объекта нет)
**up*	Только для одиночных объектов; возвращает объект, являющийся предметом владения (если такого объекта нет, последствия непредсказуемы)
up`->...`	Только для одиночных объектов; обеспечивает доступ к членам объекта, являющегося предметом владения (если такого объекта нет, последствия непредсказуемы)
up`[`*idx*`]`	Только для массивов объектов; возвращает элемент с индексом *idx* из хранимого массива (если такого объекта нет, последствия непредсказуемы)
`if (`*up*`)`	Оператор `bool()`; проверяет, является ли указатель *up* пустым
up1 `==` *up2*	Выполняет оператор `==` для хранимых указателей (возможно значение `nullptr`)
up1 `!=` *up2*	Выполняет оператор `!=` для хранимых указателей (возможно значение `nullptr`)
up1 `<` *up2*	Выполняет оператор `<` для хранимых указателей (возможно значение `nullptr`)
up1 `<=` *up2*	Выполняет оператор `<=` для хранимых указателей (возможно значение `nullptr`)

Окончание табл. 5.6

Операция	Действие
up1 > *up2*	Выполняет оператор > для хранимых указателей (возможно значение `nullptr`)
up1 >= *up2*	Выполняет оператор >= для хранимых указателей (возможно значение `nullptr`)
up`.get_deleter()`	Возвращает ссылку на оператор удаления (функциональный объект)

Конструктор, получающий в качестве аргументов указатель и функтор удаления, перегружен для нескольких разных типов.

```
D d; // экземпляр оператора удаления
unique_ptr<int, D> p1(new int, D()); // D должен соответствовать
                                     // требованиям MoveConstructible
unique_ptr<int, D> p2(new int, d);   // D должен соответствовать
                                     // требованиям CopyConstructible
unique_ptr<int, D&> p3(new int, d);  // p3 хранит ссылку на d
unique_ptr<int, const D&> p4(new int, D()); // Ошибка: rvalue-объект
                                            // оператора удаления
                                            // не может иметь ссылочный тип
                                            // оператора удаления
```

Для отдельных объектов перемещающий конструктор и оператор присваивания являются шаблонными членами, поэтому возможно преобразование типа. Все операторы сравнения и операторы удаления сделаны шаблонами для разных типов элементов.

Все операторы сравнения сравнивают простые указатели, хранящиеся в совместно используемых указателях (применяют тот же самый оператор для значений, возвращенных функцией `get()`). Все они имеют перегруженные варианты для аргумента `nullptr`. Таким образом, можно выполнить проверку, корректен ли указатель, и даже выяснить, меньше или больше простой указатель значения `nullptr`.

Специализация для массивов имеет следующие отличия от интерфейса, предусмотренного для отдельных объектов.

- Вместо операторов `*` и `->` предусмотрен оператор `[]`.
- Оператор удаления, предусмотренный по умолчанию, выполняет вызов `delete[]`, а не `delete`.
- Преобразования между разными типами не поддерживаются. В частности, нельзя ссылаться на элементы производных типов.

Отметим, что интерфейс оператора удаления отличается от соответствующего интерфейса класса `shared_ptr` (см. раздел 5.2.1). Однако, как и в случае совместно используемых указателей, оператор удаления не должен генерировать исключения.

5.2.7. Класс auto_ptr

В отличие от стандарта C++11, стандартная библиотека C++98 содержала только один класс интеллектуальных указателей, `auto_ptr`, который теперь считается устаревшим. Его целью была реализация семантики, которую в настоящее время реализует класс `unique_ptr`.

Однако класс auto_ptr создал несколько проблем.

- Во время его разработки язык еще не имел семантики перемещения для конструкторов и операторов присваивания. Однако целью класса была реализация семантики передачи владения. В результате операторы копирования и присваивания осуществляли семантику перемещения, что вызывало серьезные проблемы, особенно при передаче указателя типа auto_ptr в качестве аргумента.
- Указатели типа auto_ptr не реализовывали семантику *оператора удаления*, поэтому их можно было применять только для отдельных объектов, созданных с помощью операции new.
- Поскольку изначально указатель класса auto_ptr был единственным интеллектуальным указателем в стандартной библиотеке языка C++, он часто использовался неправильно, особенно в предположении, что он реализует семантику совместного владения, которую в настоящее время осуществляет класс shared_ptr.

Помня об опасности, которую порождает непреднамеренная потеря владения, рассмотрим следующий пример, представляющий собой наивную реализацию функции, выводящей на экран объекты, на которые ссылается указатель класса auto_ptr.

```
// это плохой пример
template <typename T>
void bad_print(std::auto_ptr<T> p) // указатель p получает владение
                                   // переданным аргументом
{
   // владеет ли указатель p объектом?
   if (p.get() == NULL) {
       std::cout << "NULL";
   }
   else {
       std::cout << *p;
   }
} // Ой, при выходе объекты, на которые
  //ссылался указатель p, были удалены
```

Каждый раз, когда указатель класса auto_ptr передается в эту реализацию функции bad_print(), объект, которым он владел (если он был), удаляется. Причина заключается в том, что владение указателя auto_ptr, который был передан как аргумент, переходит к указателю p, а он, в свою очередь, при выходе из функции удаляет объект, которым владел. Вероятно, это не входило в планы программиста и может привести к фатальным ошибкам во время выполнения программы.

```
std::auto_ptr<int> p(new int);
*p = 42;        // изменяет значение, на которое ссылается p
bad_print(p);   // ой, память, на которую ссылался p, удалена
*p = 18;        // ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ
```

Эта ситуация является типичной при передаче указателя класса auto_ptr в контейнер. При работе с указателем класса unique_ptr такая ошибка больше невозможна, потому что аргумент следует явно передавать с помощью функции std::move().

5.2.8. Заключительные замечания об интеллектуальных указателях

Как мы видели, стандарт C++11 реализует две концепции интеллектуальных указателей.

1. `shared_ptr` — для совместного владения.
2. `unique_ptr` — для исключительного владения.

Второй вариант заменяет старый класс `auto_ptr` из стандарта C++98, который теперь считается устаревшим.

Вопросы производительности

Может возникнуть вопрос: почему стандартная библиотека C++ не реализует только один класс интеллектуальных указателей с семантикой совместного владения, поскольку они также предотвращают утечку ресурсов или передачу владения? Дело в том, что совместно используемые указатели влияют на производительность программ.

Класс `shared_ptr` реализован на основе неинтрузивного (неинвазивного) подхода, подразумевающего, что объекты, управляемые этим классом, не обязаны удовлетворять специфическим требованиям, например, иметь общий базовый класс. Большим преимуществом является то, что эту концепцию можно применять к любому типу данных, включая элементарные. Цена, которую за это приходится платить, — объект класса `shared_ptr` должен содержать несколько членов: обычный указатель на объект и счетчик ссылок, ссылающихся у всех указателей на один и тот же объект. Поскольку слабые указатели могут ссылаться на совместно используемый объект, может потребоваться еще один счетчик. (Даже если на объект больше не ссылается ни один совместно используемый указатель, ссылки необходимо подсчитывать до тех пор, пока на совместно используемый объект не перестанут ссылаться слабые указатели; в противном случае нет никакой гарантии, что они вернут `0` в качестве значения функции `use_count()`.)[19]

Таким образом, совместно используемые и слабые указатели требуют наличия вспомогательных объектов, на которые ссылаются их внутренние указатели. Это исключает целую группу действий, связанных с оптимизацией (включая оптимизацию пустых базовых классов, позволяющую избежать перерасхода памяти).

Уникальные указатели ничего этого не требуют. Их "интеллектуальность" основана на специальных конструкторах и деструкторах, а также на исключении семантики копирования. При наличии пустого функтора удаления или функтора удаления, не имеющего состояния, уникальный объект занимает столько же памяти, сколько и простой указатель, причем он не требует дополнительных затрат времени по сравнению с использованием простых указателей и ручным удалением объектов. Для предотвращения нежелательных затрат времени и памяти при реализации операторов удаления следует использовать функциональные объекты (включая лямбда-функции), чтобы обеспечить максимальную оптимизацию, исключив, в идеале, дополнительные затраты вообще.

Вопросы использования

Интеллектуальные указатели не идеальны, так что при их применении следует знать, какие задачи они решают, а какие остаются нерешенными. Например, какой бы класс

[19] Благодарю Говарда Хиннанта (Howard Hinnant) за это замечание.

интеллектуальных указателей ни использовался, вы не должны создавать несколько интеллектуальных указателей для одного обычного указателя.

Пример использования совместно используемых указателей в нескольких контейнерах библиотеки STL описан в разделе 7.11.

Классы `shared_ptr` и `unique_ptr` реализуют разные подходы к работе с массивами и функторами удаления. Класс `unique_ptr` имеет специализацию для массивов, имеющую другой интерфейс. Он более гибкий и меньше влияет на производительность программ, но требует более внимательной работы.

В заключение отметим, что интеллектуальные указатели не являются потокобезопасными, хотя и предоставляют некоторые гарантии. Подробности изложены в разделе 5.2.4.

5.3. Числовые пределы

Числовые типы имеют платформно-зависимые пределы. Стандартная библиотека языка C++ задает эти пределы в шаблоне `numeric_limits`. Эти числовые пределы заменяют и дополняют обычные константы препроцессора языка C, которые по-прежнему доступны для целочисленных типов в заголовках `<climits>` и `<limits.h>` и для чисел с плавающей точкой в заголовках `<cfloat>` и `<float.h>`. Новая концепция числовых пределов имеет свои преимущества. Во-первых, она повышает безопасность с точки зрения типов, во-вторых, позволяет программисту писать шаблонные функции, вычисляющие эти пределы.

Числовые пределы обсуждаются в оставшейся части раздела. Однако следует отметить, что всегда лучше писать платформно-независимый код, используя минимальную гарантированную точность для типов. Эти минимальные значения указаны в табл. 5.7.[20]

Таблица 5.7. Минимальный размер встроенных типов

Тип	Минимальный размер
`char`	1 байт (8 битов)
`short int`	2 байта
`int`	2 байта
`long int`	4 байта
`long long int`	8 байтов
`float`	4 байта
`double`	8 байтов
`long double`	8 байтов

Класс `numeric_limits<>`

Обычно шаблоны используются для реализации каких-то операций, которые могут применяться к любому типу. Однако шаблоны также можно использовать для реализации общего интерфейса для каждого типа, что бывает очень полезным. Для этого можно

[20] Обратите внимание на то, что здесь байт — это октет из 8 битов. Строго говоря, тип `long int` может состоять из байта, содержащего не менее 32 битов.

предусмотреть специализации общего шаблона. Типичный пример такого приема — класс numeric_limits, работающий следующим образом.

- Общий шаблон обеспечивает числовые значения, заданные по умолчанию для любого типа.

```
namespace std {
   // общие числовые пределы, заданные по умолчанию для любого типа
   template <typename T>
   class numeric_limits {
      public:
        // по умолчанию для любого типа T специализация не предусмотрена
        static constexpr bool is_specialized = false;
        ... // другие члены для общего шаблона не нужны
   };
}
```

- Этот общий шаблон для числовых пределов означает, что для типа T числовых пределов нет. Для этого член is_specialized задается равным false.
- Специализации шаблона определяют числовые пределы для каждого числового типа.

```
namespace std {
   // числовые пределы для типа int
   // - определенные реализацией
   template<> class numeric_limits<int> {
      public:
         // да, специализация числовых пределов для типа int существует
         static constexpr bool is_specialized = true;

         static constexpr int min() noexcept {
            return -2147483648;
         }
         static constexpr int max() noexcept {
            return 2147483647;
         }
         static constexpr int digits = 31;
         ...
   };
}
```

Здесь член is_specialized установлен равным true, а все остальные члены класса имеют значения числовых пределов для конкретного типа.

Общий шаблонный класс numeric_limits и его стандартные специализации определены в заголовке <limits>. Специализации предусмотрены для любого элементарного типа, который может представлять числовые значения: bool, char, signed char, unsigned char, char16_t, char32_t, wchar_t, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float, double и long double.[21] Их легко дополнить пользовательскими числовыми типами.

[21] Специализации для типов char16_t, char32_t, long long и unsigned long long предусмотрены только в стандарте C++11.

Все члены класса numeric_limits<> и их описание приведены в табл. 5.8. В ней также указаны соответствующие константы языка С, определенные в заголовках <climits>, <limits.h>, <cfloat> и <float.h>.

Таблица 5.8. Члены класса numeric_limits<>

Член	Предназначение	Константы языка С
is_specialized	Тип имеет специализацию для числовых пределов	
is_signed	Тип имеет знак	
is_integer	Тип является целочисленным	
is_exact	Вычисления не сопровождаются ошибками округления (true для всех целочисленных типов)	
is_bounded	Набор значений конечен (true для всех встроенных типов)	
is_modulo	Сумма двух положительных чисел может оказаться меньше каждого из них	
is_iec559	Соответствует стандартам IEC 559 и IEEE 754	
min()	Минимальное конечное значение (минимальное нормализованное значение для чисел с плавающей точкой с денормализацией; имеет смысл, если is_bounded||!is_signed)	INT_MIN, FLT_MIN, CHAR_MIN, ...
max()	Максимальное конечное число (имеет смысл, если is_bounded)	INT_MAX, FLT_MAX, ...
lowest()	Максимальное отрицательное конечное число (имеет смысл, если is_bounded; только для стандарта С++11)	
digits	Символ/целое число: количество битов, исключая знаковые (двоичные цифры) Число с плавающей точкой: количество цифр в мантиссе в системе счисления с основанием radix	CHAR_BIT FLT_MANT_DIG, ...
digits10	Количество десятичных знаков (имеет смысл, если is_bounded)	FLT_DIG, ...
max_digits10	Количество требуемых десятичных знаков, чтобы всегда можно было различить разные значения (имеет смысл для всех типов чисел с плавающей точкой; только для стандарта С++11)	
radix	Целое число: база системы счисления представления (практически всегда 2). Плавающая точка: база экспоненциального представления	FLT_RADIX
min_exponent	Минимальная целая степень в системе счисления с основанием radix	FLT_MIN_EXP, ...

Окончание табл. 5.8

Член	Предназначение	Константы языка C
`max_exponent`	Максимальная положительная целая степень в системе счисления с основанием `radix`	`FLT_MAX_EXP,...`
`min_exponent10`	Минимальная отрицательная целая степень в системе счисления с основанием 10	`FLT_MIN_10_EXP,...`
`max_exponent10`	Максимальная положительная целая степень в системе счисления с основанием 10	`FLT_MAX_10_EXP,...`
`epsilon()`	Разница между `1` и наименьшим значением, которое больше `1`	`FLT_EPSILON,...`
`round_style`	Способ округления	
`round_error()`	Величина максимальной ошибки округления (в соответствие со стандартом ISO/IEC 10967-1)	
`has_infinity`	Тип имеет представление для положительной бесконечности	
`infinity()`	Представление положительной бесконечности, если такое предусмотрено	
`has_quiet_NaN`	Тип имеет представление для не сигнализирующего "не числа"	
`quiet_NaN()`	Представление для не сигнализирующего "не числа", если такое предусмотрено	
`has_signaling_NaN`	Тип имеет представление для сигнализирующего "не числа"	
`signaling_NaN()`	Представление для сигнализирующего "не числа", если такое предусмотрено	
`has_denorm`	Допускает ли тип денормализованные значения (переменное количество битов в показателе степени)	
`has_denorm_loss`	Потеря точности распознается как потеря денормализации, а не как неточный результат	
`denorm_min()`	Минимальное положительное денормализованное значение	
`traps`	Реализована обработка прерывания	
`tinyness_before`	Перед округлением определяется малость числа	

Ниже приведена полная специализация числовых пределов для типа `float`, зависящая от платформы и демонстрирующая точные сигнатуры функций-членов.

```
namespace std {
   template<> class numeric_limits<float> {
      public:
        // да, специализация для числовых пределов
        // типа float существует
        static constexpr bool is_specialized = true;
```

```
    inline constexpr float min() noexcept {
       return 1.17549435E-38F;
    }

    inline constexpr float max() noexcept {
       return 3.40282347E+38F;
    }

    inline constexpr float lowest() noexcept {
       return -3.40282347E+38F;
    }

    static constexpr int digits = 24;
    static constexpr int digits10 = 6;
    static constexpr int max_digits10 = 9;

    static constexpr bool is_signed = true;
    static constexpr bool is_integer = false;
    static constexpr bool is_exact = false;
    static constexpr bool is_bounded = true;
    static constexpr bool is_modulo = false;
    static constexpr bool is_iec559 = true;
    static constexpr int radix = 2;

    inline constexpr float epsilon() noexcept {
       return 1.19209290E-07F;
    }

    static constexpr float_round_style round_style
        = round_to_nearest;
    inline constexpr float round_error() noexcept {
        return 0.5F;
    }

    static constexpr int min_exponent = -125;
    static constexpr int max_exponent = +128;
    static constexpr int min_exponent10 = -37;
    static constexpr int max_exponent10 = +38;

    static constexpr bool has_infinity = true;
    inline constexpr float infinity() noexcept { return ...; }
    static constexpr bool has_quiet_NaN = true;
    inline constexpr float quiet_NaN() noexcept { return ...; }
    static constexpr bool has_signaling_NaN = true;
    inline constexpr float signaling_NaN() noexcept { return ...; }
    static constexpr float_denorm_style has_denorm = denorm_absent;
    static constexpr bool has_denorm_loss = false;
    inline constexpr float denorm_min() noexcept { return min(); }

    static constexpr bool traps = true;
    static constexpr bool tinyness_before = true;
};
}
```

Отметим, что в стандарте C++11 все члены объявлены как `constexpr` (см. раздел 3.1.8). Например, функцию `max()` можно использовать там, где требуются выражения времени компиляции.

```
static const int ERROR_VALUE = std::numeric_limits<int>::max();
float a[std::numeric_limits<short>::max()];
```

До появления стандарта C++11 все члены класса, не являющиеся функциями, были константными и статическими, поэтому их значения определялись на этапе компиляции. Однако функции-члены были только `static`, так что вышеприведенные выражения были невозможны. Отметим также, что до появления стандарта C++11 функции `lowest()` и константы `max_digits10` не было, а вместо модификатора `nonexcept` использовались пустые спецификации исключений (см. раздел 3.1.7).

Значения члена `round_style` приведены в табл. 5.9, значения члена `has_denorm` — в табл. 5.10. К сожалению, член `has_denorm` не называется `denorm_style`. Это произошло потому, что в процессе стандартизации было поздно изменять булев тип на значение перечисления. Однако член `has_denorm` можно использовать как булево значение, потому что стандарт гарантирует, что значение `denorm_absent` равно `0`, что эквивалентно `false`, в то время как значение `denorm_present` равно `1`, а `denorm_indeterminate` равно `-1`, что в обоих случаях эквивалентно значению `true`. Таким образом, член `has_denorm` можно рассматривать как булев индикатор того, что тип допускает денормализованные значения.

Таблица 5.9. Способ округления в классе `numeric_limits<>`

Способ округления	Смысл
`round_toward_zero`	Округление для нуля
`round_to_nearest`	Округление до ближайшего представимого числа
`round_toward_infinity`	Округление до положительной бесконечности
`round_toward_neg_infinity`	Округление до отрицательной бесконечности
`round_indeterminate`	Неопределимый

Таблица 5.10. Способ денормализации в классе `numeric_limits<>`

Способ денормализации	Смысл
`denorm_absent`	Тип не допускает денормализованные значения
`denorm_present`	Тип допускает денормализацию значения до ближайшего представимого числа
`denorm_indeterminate`	Неопределимый

Пример использования класса `numeric_limits<>`

Следующий пример демонстрирует возможное использование некоторых числовых пределов, например, максимальных значений для определенных типов и выяснение, есть ли знак у типа `char`.

```
// util/limits1.cpp

#include <iostream>
```

```
#include <limits>
#include <string>
using namespace std;

int main()
{
   // используется текстуальное представление булевских значений
   cout << boolalpha;

   // вывод максимального значения для целочисленных типов
   cout << "max(short): " << numeric_limits<short>::max() << endl;
   cout << "max(int): " << numeric_limits<int>::max() << endl;
   cout << "max(long): " << numeric_limits<long>::max() << endl;
   cout << endl;

   // вывод максимального значения для чисел с плавающей точкой
   cout << "max(float): "
        << numeric_limits<float>::max() << endl;
   cout << "max(double): "
        << numeric_limits<double>::max() << endl;
   cout << "max(long double): "
        << numeric_limits<long double>::max() << endl;
   cout << endl;

   // вывод, является ли тип char знаковым
   cout << "is_signed(char): "
        << numeric_limits<char>::is_signed << endl;
   cout << endl;

   // вывод, существуют ли
   // числовые пределы для типа string
   cout << "is_specialized(string): "
        << numeric_limits<string>::is_specialized << endl;
}
```

Вывод этой программы зависит от платформы. Вот один из возможных вариантов вывода:

```
max(short): 32767
max(int):   2147483647
max(long):  2147483647

max(float):       3.40282e+38
max(double):      1.79769e+308
max(long double): 1.79769e+308

is_signed(char): false

is_specialized(string): false
```

Последняя строка показывает, что для типа `string` числовые пределы не определены. Это логично, потому что строки не являются числовыми значениями. Однако этот пример показывает, что такую проверку можно выполнить для любого типа.

5.4. Свойства и утилиты типов

Почти все компоненты стандартной библиотеки языка C++ основаны на шаблонах. Для поддержки шаблонного программирования, которое иногда называют *метапрограммированием*, в библиотеке предусмотрены шаблонные утилиты, предназначенные как для программистов, так и для разработчиков библиотек.

Свойства типов (type traits), введенные в документе TR1 и расширенные в стандарте C++11, являются основой механизма для определения поведения, зависящего от типов. Их можно использовать для оптимизации кода, описывающего типы с особыми возможностями.

Могут оказаться полезными и другие утилиты, такие как оболочки для ссылок и функций.

5.4.1. Предназначение свойств типов

Свойство типа обеспечивает способ для работы с его атрибутами. Это шаблон, который на этапе компиляции создает конкретный тип или значение на основе одного или нескольких шаблонных аргументов, которые обычно являются типами.

Рассмотрим следующий пример:

```
template <typename T>
void foo (const T& val)
{
   if (std::is_pointer<T>::value) {
       std::cout << "foo() вызвана для указателя" << std::endl;
   }
   else {
       std::cout << "foo() вызвана для значения" << std::endl;
   }
   ...
}
```

Здесь свойство `std::is_pointer`, определенное в `<type_traits>`, используется для проверки того, является ли тип `T` типом указателя. Фактически шаблонный класс `is_pointer<>` дает либо тип `true_type`, либо тип `false_type`, для которых значение `::value` равно либо `true`, либо `false`. В результате, если полученный функцией `foo()` параметр `val` является указателем, она выведет строку

```
foo() вызвана для указателя
```

Отметим, однако, что следующий код является неправильным:

```
template <typename T>
void foo (const T& val)
{
   std::cout << (std::is_pointer<T>::value ? *val : val)
             << std::endl;
}
```

Причина заключается в том, что этот код генерируется как для `*val`, так и для `val`. Даже при передаче типа `int`, для которого выражение `is_pointer<T>::value` на этапе компиляции выдает значение `false`, код разворачивается в следующий оператор:

```
cout << (false ? *val : val) << endl;
```

Эта конструкция не компилируется, потому что выражение `*val` для типа `int` является некорректным.

Однако можно написать следующий код:

```
// реализация функции foo() для типов указателей:
template <typename T>
void foo_impl (const T& val, std::true_type)
{
   std::cout << "foo() вызвана для указателя на " << *val
             << std::endl;
}

// реализация функции foo() для типов не-указателей:
template <typename T>
void foo_impl (const T& val, std::false_type)
{
   std::cout << "foo() вызвана для значения " << val
             << std::endl;
}

template <typename T>
void foo (const T& val)
{
   foo_impl (val, std::is_pointer<T>());
}
```

Здесь в функции `foo()` выражение

```
std::is_pointer<T>()
```

во время компиляции дает тип `std::true_type` или `std::false_type`, который и определяет, какой из перегруженных вариантов функции `foo_impl()` будет конкретизирован.

Почему это лучше, чем два перегруженных варианта для функции `foo()`: один для обычных типов, а другой — для типов указателей?

```
template <typename T>
void foo (const T& val);      // общая реализация

template <typename T>
void foo<T*> (const T& val); // частичная специализация для указателей
```

Один из возможных ответов заключается в том, что иногда приходится создавать слишком много вариантов перегрузки. В целом мощь свойств типов обусловлена тем фактом, что они являются строительными конструкциями обобщенного кода. Этот факт можно продемонстрировать двумя примерами.

Гибкая перегрузка целочисленных типов

В работе [Becker:LibExt] Пете Беккер (Pete Becker) привел прекрасный пример, который мы слегка модифицировали. Допустим, у нас есть функция `foo()`, которую для аргументов, являющихся целыми числами и числами с плавающей точкой, необходимо реализовать по-разному. Обычно в таких случаях функцию перегружают для всех возможных типов целых чисел и чисел с плавающей точкой[22].

```
void foo (short);            // целочисленная версия
void foo (unsigned short);
void foo (int);
...
void foo (float);           // версия для плавающей точки
void foo (double);
void foo (long double);
```

Это повторение не только утомительно, но и создает проблему, которая заключается в том, что для новых целочисленных типов и типов чисел с плавающей точкой данный код может не работать, независимо от того, предусмотрены ли они стандартом, как `long long`, или пользователем.

С помощью свойств типа можно написать следующий код:

```
template <typename T>
void foo_impl (T val, true_type);     // версия для целочисленных типов

template <typename T>
void foo_impl (T val, false_type);    // версия для плавающей точки

template <typename T>
void foo (T val)
{
   foo_impl (val, std::is_integral<T>());
}
```

Таким образом, мы имеем две реализации — одну для целочисленных типов, а другую для типов чисел с плавающей точкой — и выбираем правильную реализацию в зависимости от того, какой тип создает шаблонный класс `std::is_integral<>`.

Обработка общего типа

Другим примером, демонстрирующим полезность свойств типов, является потребность в обработке общего типа для двух или более типов. В этом случае значения двух разных типов можно рассматривать как значения общего типа. Например, таким типом может быть тип минимального из двух значений или тип суммы двух значений разных типов. Если не предусмотреть такой тип, то при реализации функции, определяющей минимальное из двух значений разных типов, становится непонятным, каким должен быть тип возвращаемого значения.

---

[22] В соответствии со стандартом языка C++ термин *целочисленный тип* (integral type) относится также к типам `bool` и символьным типам, но в этом примере это не имеет значения.

```
template <typename T1, typename T2>
??? min (const T1& x, const T2& y);
```

Используя свойства типов, при объявлении общего типа можно просто использовать класс `std::common_type<>`.

```
template <typename T1, typename T2>
typename std::common_type<T1,T2>::type min (const T1& x, const T2& y);
```

Например, выражение `std::common_type<T1,T2>::type` выдает `int`, если оба аргумента имеют типы `int`; `long`, если один из них имеет тип `int`, а другой `long`; или `std::string`, если один из них является строкой, а второй — строковым литералом (тип `const char*`).

Как это работает? Здесь просто используется правило, реализованное для оператора `?:`, который возвращает тип результата в зависимости от типа обоих операндов. Фактически класс `std::common_type<>` реализован следующим образом:

```
template <typename T1, typename T2>
struct common_type<T1,T2> {
   typedef decltype(true ? declval<T1>() : declval<T2>()) type;
};
```

где `decltype` — новое ключевое слово в стандарте C++11 (см. раздел 3.1.11), предназначенное для получения типа выражения, а `declval<>` — вспомогательное свойство для предоставления объявленного значения переданного типа без его вычисления (генерации для него rvalue-ссылки).

Таким образом, если оператор `?:` может определить общий тип, класс `common_type<>` выдает его. Если нет, можно предусмотреть перегрузку `common_type<>` (именно этот прием использован в заголовке `<chrono>` для комбинирования продолжительности времени; см. раздел 5.7.2).

5.4.2. Подробное описание свойств типов

Свойства типов обычно определяются в заголовочном файле `<type_traits>`.

(Унарные) предикаты типов

Как указывалось в разделе 5.4.1, предикаты типов дают тип `std::true_type`, если заданное свойство имеет место, и `std::false_type`, если нет. Эти типы являются специализациями вспомогательного класса `std::integral_constant`, поэтому их соответствующие значения члена `value` равны `true` или `false`.

```
namespace std {
   template <typename T, T val>
   struct integral_constant {
      static constexpr T value = val;
      typedef T value_type;
      typedef integral_constant<T,val> type;
      constexpr operator value_type() {
         return value;
      }
```

```
    };
    typedef integral_constant<bool,true> true_type;
    typedef integral_constant<bool,false> false_type;
}
```

Предикаты для всех типов приведены в табл. 5.11, а свойства, уточняющие детали классов, — в табл. 5.12.

Таблица 5.11. Свойства для проверки характеристик типов

Свойство	Действие
`is_void<T>`	Тип `void`
`is_integral<T>`	Целочисленный тип (включая `bool`, `char`, `char16_t`, `char32_t`, `wchar_t`)
`is_floating_point<T>`	Тип чисел с плавающей точкой (`float`, `double`, `long double`)
`is_arithmetic<T>`	Целочисленный тип (включая `bool` и символы) или тип числа с плавающей точкой
`is_signed<T>`	Арифметический тип со знаком
`is_unsigned<T>`	Арифметический тип без знака
`is_const<T>`	Квалифицированный модификатором `const`
`is_volatile<T>`	Квалифицированный модификатором `volatile`
`is_array<T>`	Обычный массив (не тип `std::array`)
`is_enum<T>`	Перечисление
`is_union<T>`	Объединение
`is_class<T>`	Класс/структура, но не объединение
`is_function<T>`	Функция
`is_reference<T>`	Lvalue- или rvalue-ссылка
`is_lvalue_reference<T>`	Lvalue-ссылка
`is_rvalue_reference<T>`	Rvalue-ссылка
`is_pointer<T>`	Указатель (включая указатель на функцию, но не указатель на нестатическую функцию-член)
`is_member_pointer<T>`	Указатель на нестатическую функцию-член
`is_member_object_pointer<T>`	Указатель на нестатический член-данные
`is_member_function_pointer<T>`	Указатель на нестатическую функцию-член
`is_fundamental<T>`	`void`, целочисленный (включая `bool` и символы), числа с плавающей точкой или `std::nullptr_t`
`is_scalar<T>`	Целочисленный (включая `bool` и символы), числа с плавающей точкой, перечисление, указатель, указатель на член, `std::nullptr_t`

Окончание табл. 5.11

Свойство	Действие
`is_object<T>`	Любой тип, за исключением `void`, функции или ссылки
`is_compound<T>`	Массив, перечисление, объединение, класс, функция, ссылка или указатель
`is_trivial<T>`	Скаляр, тривиальный класс или массивы объектов таких типов
`is_trivially_copyable<T>`	Скаляр, тривиально копируемый класс или массивы объектов таких типов
`is_standard_layout<T>`	Скаляр, стандартный макетный класс (standard layout class) или массивы объектов этих типов
`is_pod<T>`	Простой тип данных (тип, для которого копирование объектов выполняет функция `memcpy()`)
`is_literal_type<T>`	Скаляр, ссылка, класс или массивы объектов таких типов

Таблица 5.12. Свойства для проверки атрибутов классов

Свойство	Действие
`is_empty<T>`	Класс без членов, виртуальные функции-члены или виртуальные базовые классы
`is_polymorphic<T>`	Класс с (производной) виртуальной функцией-членом
`is_abstract<T>`	Абстрактный класс (класс, содержащий по крайней мере одну чисто виртуальную функцию)
`has_virtual_destructor<T>`	Класс с виртуальным деструктором
`is_default_constructible<T>`	Класс, допускающий конструктор по умолчанию
`is_copy_constructible<T>`	Класс, допускающий копирующий конструктор
`is_move_constructible<T>`	Класс, допускающий перемещающий конструктор
`is_copy_assignable<T>`	Класс, допускающий копирующее присваивание
`is_move_assignable<T>`	Класс, допускающий перемещающее присваивание
`is_destructible<T>`	Класс с вызываемым деструктором (не удаленным, не защищенным и не закрытым)
`is_trivially_default_constructible<T>`	Класс, допускающий тривиальный конструктор по умолчанию
`is_trivially_copy_constructible<T>`	Класс, допускающий тривиальный копирующий конструктор по умолчанию
`is_trivially_move_constructible<T>`	Класс, допускающий тривиальный перемещающий конструктор
`is_trivially_copy_assignable<T>`	Класс, допускающий тривиальное копирующее присваивание

Окончание табл. 5.12

Свойство	Действие
`is_trivially_move_assignable<T>`	Класс с тривиальным перемещающим присваиванием
`is_trivially_destructible<T>`	Класс с тривиальным вызываемым деструктором
`is_nothrow_default_constructible<T>`	Класс, допускающий конструктор по умолчанию, который не генерирует исключения
`is_nothrow_copy_constructible<T>`	Класс, допускающий копирующий конструктор по умолчанию, который не генерирует исключения
`is_nothrow_move_constructible<T>`	Класс, допускающий перемещающий конструктор, который не генерирует исключения
`is_nothrow_copy_assignable<T>`	Класс, допускающий копирующее присваивание, не генерирующее исключение
`is_nothrow_move_assignable<T>`	Класс, допускающий перемещающее присваивание, не генерирующее исключение
`is_nothrow_destructible<T>`	Класс, допускающий вызываемый деструктор, не генерирующий исключение

Тип `bool` и все символьные типы (`char`, `char16_t`, `char32_t` и `wchar_t`) относятся к целочисленным типам, а тип `std::nullptr_t` (см. раздел 3.1.1) считается элементарным.

В большинстве (хотя и не все) свойства являются унарными. Иначе говоря, они используют только один шаблонный аргумент. Например, предикат `is_const<>` проверяет, является ли передаваемый тип константным.

```
is_const<int>::value                  // false
is_const<const volatile int>::value   // true
is_const<int* const>::value           // true
is_const<const int*>::value           // false
is_const<const int&>::value           // false
is_const<int[3]>::value               // false
is_const<const int[3]>::value         // true
is_const<int[]>::value                // false
is_const<const int[]>::value          // true
```

Отметим, что неконстантный указатель или ссылка на константный тип являются неконстантными, в то время как обычный массив константных элементов считается константным[23].

Свойства, проверяющие семантику копирования и перемещения, проверяют только то, возможны ли соответствующие выражения. Например, тип с копирующим конструктором, имеющим константный аргумент, но не имеющий перемещающего конструктора, все же имеет семантику перемещения.

Свойство типа `is_nothrow...` особенно удобно для формулировки спецификаций `noexcept` (см. раздел 3.1.7).

[23] Корректность этого утверждения в настоящее время является предметом дискуссий в группе по стандартизации ядра языка.

Свойства отношений между типами

В табл. 5.13 перечислены свойства типа, допускающие проверку отношений между типами. В частности, можно проверить, предусмотрены ли конструкторы и операторы присваивания в классе.

Таблица 5.13. Свойства для проверки отношений между типами

Свойство	Действие
`is_same<`*T1*`,`*T2*`>`	Типы *T1* и *T2* являются одинаковыми (с учетом квалификаторов `const`/`volatile`)
`is_base_of<`*T*`,`*D*`>`	Тип *T* является базовым классом по отношению к типу *D*
`is_convertible<`*T*`,`*T2*`>`	Тип *T* может быть преобразован в тип *T2*
`is_constructible<`*T*`,`*Args*`...>`	Можно инициализировать тип *T* типами *Args*
`is_trivially_constructible<`*T*`,`*Args*`...>`	Можно тривиально инициализировать тип *T* типами *Args*
`is_nothrow_constructible<`*T*`,`*Args*`...>`	Инициализация типа *T* типами *Args* не генерирует исключений
`is_assignable<`*T*`,`*T2*`>`	Можно присваивать тип *T2* типу *T*
`is_trivially_assignable<`*T*`,`*T2*`>`	Можно тривиально присваивать тип *T2* типу *T*
`is_nothrow_assignable<`*T*`,`*T2*`>`	Присваивание типа *T2* типу *T* не генерирует исключений
`uses_allocator<`*T*`,`*Alloc*`>`	Тип *Alloc* может быть преобразован в тип *T*`::allocator_type`

Отметим, что тип наподобие `int` допускает как lvalue-, так и rvalue-значения. Поскольку присваивание

```
42 = 77;
```

невозможно, предикат `is_assignable<>` для типов, имеющих первый шаблонный аргумент, не являющийся классом, всегда возвращает значение `false_type`. Однако если первым шаблонным аргументом является класс, то этот предикат возвращает истинное значение, потому что существует старое правило, позволяющее вызывать функции-члены из rvalue-значений классов[24]. Например:

```
is_assignable<int,int>::value                    // false
is_assignable<int&,int>::value                   // true
is_assignable<int&&,int>::value                  // false
is_assignable<long&,int>::value                  // true
is_assignable<int&,void*>::value                 // false
is_assignable<void*,int>::value                  // false
is_assignable<const char*,std::string>::value    // false
is_assignable<std::string,const char*>::value    // true
```

[24] Благодарю за это замечание Даниэля Крюглера.

Рассмотрим примеры применения свойства `is_constructible<>`:

```
is_constructible<int>::value                              // true
is_constructible<int,int>::value                          // true
is_constructible<long,int>::value                         // true
is_constructible<int,void*>::value                        // false
is_constructible<void*,int>::value                        // false
is_constructible<const char*,std::string>::value          // false
is_constructible<std::string,const char*>::value          // true
is_constructible<std::string,const char*,int,int>::value // true
```

Свойство `std::uses_allocator<>` определено в заголовке `<memory>` (см. раздел 19.1).

Модификаторы типа

Свойства, перечисленные в табл. 5.14, позволяют модифицировать типы.

Все модифицирующие свойства добавляют атрибут типа, которого в нем не было, или удаляют ранее существовавший атрибут. Например, тип `int` можно лишь расширить:

```
typedef int T;
add_const<T>::type            // const int
add_lvalue_reference<T>::type // int&
add_rvalue_reference<T>::type // int&&
add_pointer<T>::type          // int*
make_signed<T>::type          // int
make_unsigned<T>::type        // unsigned int
remove_const<T>::type         // int
remove_reference<T>::type     // int
remove_pointer<T>::type       // int
```

а тип `const int&` можно как сузить, так и расширить:

```
typedef const int& T;
add_const<T>::type            // const int&
add_lvalue_reference<T>::type // const int&
add_rvalue_reference<T>::type // const int& (да, lvalue остается lvalue)
add_pointer<T>::type          // const int*
make_signed<T>::type          // непредсказуемое поведение
make_unsigned<T>::type        // непредсказуемое поведение
remove_const<T>::type         // const int&
remove_reference<T>::type     // const int
remove_pointer<T>::type       // const int&
```

Напомним, что ссылка на константный тип не является константным типом, поэтому у него нельзя отнять модификатор `const`. Отметим, что предикат `add_pointer<>` подразумевает применение предиката `remove_reference<>`. Однако предикаты `make_signed<>` и `make_unsigned<>` требуют, чтобы аргументы были либо целочисленными, либо перечислениями, исключая `bool`, поэтому передача ссылок приводит к непредсказуемым последствиям.

Таблица 5.14. Свойства для модификации типов

Свойство	Действие
`remove_const<T>`	Соответствующий тип без `const`
`remove_volatile<T>`	Соответствующий тип без `volatile`
`remove_cv<T>`	Соответствующий тип без `const` и `volatile`
`add_const<T>`	Соответствующий тип `const`
`add_volatile<T>`	Соответствующий тип `volatile`
`add_cv<T>`	Соответствующий тип `const volatile`
`make_signed<T>`	Соответствующий тип со знаком, не являющийся ссылочным
`make_unsigned<T>`	Соответствующий тип без знака, не являющийся ссылочным
`remove_reference<T>`	Соответствующий тип, не являющийся ссылочным
`add_lvalue_reference<T>`	Соответствующий ссылочный lvalue-тип (rvalue-тип становится lvalue-типом)
`add_rvalue_reference<T>`	Соответствующий ссылочный rvalue-тип (lvalue-тип остается lvalue-типом)
`remove_pointer<T>`	Тип объекта, на который ссылается указатель (если предикат применяется не к указателю, тип не изменяется)
`add_pointer<T>`	Тип указателя для соответствующего типа, не являющегося ссылочным

Отметим, что свойство `add_lvalue_reference<>` преобразует rvalue-ссылку в lvalue-ссылку, а `add_rvalue_reference<>` *не* преобразует lvalue-ссылку в rvalue-ссылку (тип остается неизменным). Следовательно, чтобы конвертировать lvalue-ссылку в rvalue-ссылку, необходимо вызвать

```
add_rvalue_reference<remove_reference<T>::type>::type
```

Другие свойства типов

Остальные свойства типов приведены в табл. 5.15. К ним относятся запросы для специальных свойств, проверки отношений между типами и более сложные преобразования типов.

Таблица 5.15. Другие свойства типов

Свойства	Действие
`rank<T>`	Количество измерений массива типа *T* (или 0)
`extent<T, I=0>`	Расширение размерности *I* (или 0)
`remove_extent<T>`	Тип элементов массива (в противном случае — тот же самый тип)
`remove_all_extents<T>`	Тип элементов для многомерных массивов (в противном случае — тот же самый тип)
`underlying_type<T>`	Тип, лежащий в основе перечисления (см. раздел 3.1.13)

Окончание табл. 5.15

Свойства	Действие
`decay<T>`	Преобразование в соответствующий тип "по значению"
`enable_if<B,T=void>`	Возвращает тип *T*, только если значение *B* типа `bool` равно `true`
`conditional<B,T,F>`	Возвращает тип *T*, если значение *B* типа `bool` равно `true`, в противном случае возвращает тип *F*
`common_type<T1,...>`	Общий тип для всех передаваемых типов
`result_of<F,ArgTypes>`	Тип результата вызова *F* с аргументами *ArgTypes*
`alignment_of<T>`	Эквивалент `alignof(T)`
`aligned_storage<Len>`	Тип *Len* байтов с выравниванием по умолчанию
`aligned_storage<Len,Align>`	Тип *Len* байтов, выровненных в соответствии с делителем *Align* типа `size_t`
`aligned_union<Len,Types...>`	Тип *Len* байтов, выровненный для объединения *Types...*

Свойства, работающие с предикатами `rank` и `extent`, позволяют обрабатывать (многомерные) массивы. Например:

```
rank<int>::value                    // 0
rank<int[]>::value                  // 1
rank<int[5]>::value                 // 1
rank<int[][7]>::value               // 2
rank<int[5][7]>::value              // 2
extent<int>::value                  // 0
extent<int[]>::value                // 0
extent<int[5]>::value               // 5
extent<int[][7]>::value             // 0
extent<int[5][7]>::value            // 5
extent<int[][7],1>::value           // 7
extent<int[5][7],1>::value          // 7
extent<int[5][7],2>::value          // 0
remove_extent<int>::type            // int
remove_extent<int[]>::type          // int
remove_extent<int[5]>::type         // int
remove_extent<int[][7]>::type       // int[7]
remove_extent<int[5][7]>::type      // int[7]
remove_all_extents<int>::type       // int
remove_all_extents<int[]>::type     // int
remove_all_extents<int[5]>::type    // int
remove_all_extents<int[][7]>::type  // int
remove_all_extents<int[5][7]>::type // int
```

Свойство `decay<>` предоставляет возможность конвертировать тип `T` в соответствующий ему тип, когда этот тип передается по значению. Таким образом, оно конвертирует типы массивов и функций в указатели, а также lvalue-значения в rvalue-значения, включая удаление модификаторов `const` и `volatile`. Примеры их использования см. в разделе 5.1.1.

Как указано в разделе 5.4.1, свойство `common_type<>` возвращает общий тип для всех передаваемых типов (можно передавать один, два аргумента и больше).

5.4.3. Обертки для ссылок

Класс `std::reference_wrapper<>`, объявленный в заголовке `<functional>`, используется в основном для передачи ссылок в шаблонные функции, которые принимают параметры по значению. Для заданного типа `T` этот класс предоставляет функцию `ref()` для неявного преобразования в `T&` и `cref()` для неявного преобразования в `const T&`. Это обычно позволяет шаблонным функциям работать со ссылками без специализации.

Например, после объявления

```
template <typename T>
void foo (T val);
```

в вызове

```
int x;
foo (std::ref(x));
```

тип `T` становится `int&`, а в вызове

```
int x;
foo (std::cref(x));
```

тип `T` становится `const int&`.

Эта функциональная возможность используется в разных местах стандартной библиотекой языка C++. Соответствующие примеры приведены ниже.

- Функция `make_pair()` использует ее для того, чтобы создавать объекты класса `pair<>`, состоящие из ссылок (см. раздел 5.1.1).
- Функция `make_tuple()` использует ее для того, чтобы создавать объекты класса `tuple<>`, состоящие из ссылок (см. раздел 5.1.2).
- Связыватели используют ее для связывания ссылок (см. раздел 10.2.2).
- Потоки используют ее для передачи аргументов по ссылке (см. раздел 18.2.2).

Отметим также, что класс `reference_wrapper` позволяет использовать ссылки в качестве объектов первого класса, таких как тип элементов массива или контейнера STL.

```
std::vector<MyClass&> coll;                          // Ошибка
std::vector<std::reference_wrapper<MyClass>> coll; // OK
```

Детали описаны в разделе 7.11.

5.4.4. Обертки функциональных типов

Класс `std::function<>`, объявленный в заголовке `<functional>`, предоставляет полиморфные обертки, обобщающие понятие указателя на функцию. Этот класс позволяет использовать *вызываемые объекты* (функции, функции-члены, функторы и лямбда-функции; см. раздел 4.4) в качестве объектов первого класса. Например:

```
void func (int x, int y);

// инициализация коллекции заданий:
```

```
std::vector<std::function<void(int,int)>> tasks;
tasks.push_back(func);
tasks.push_back([] (int x, int y) {
                    ...
                    });
// вызов каждой задачи:
for (std::function<void(int,int)> f : tasks) {
    f(33,66);
}
```

Для вызова каждой задачи можно также просто выполнить следующий код.

```
// вызов каждой задачи:
for (auto f : tasks) {
  f(33,66);
}
```

При использовании функций-членов объект, из которого они вызываются, должен передаваться как первый аргумент.

```
class C {
   public:
      void memfunc (int x, int y) const;
};

std::function<void(const C&,int,int)> mf;
mf = &C::memfunc;
mf(C(),42,77);
```

Другое применение этой функциональной возможности — объявление функций, возвращающих лямбда-функции (см. раздел 3.1.10).

Отметим, что вызов функции без указания *цели* (т.е. конкретного вызываемого объекта) приводит к генерированию исключения `std::bad_function_call` (см. раздел 4.3.1).

```
std::function<void(int,int)> f;
f(33,66); // генерирование исключения std::bad_function_call
```

5.5. Вспомогательные функции

Стандартная библиотека языка C++ содержит несколько вспомогательных функций, например, вычисляющих минимум и максимум, обменивающих значения и предоставляющих дополнительные операторы сравнения.

5.5.1. Вычисление минимума и максимума

В табл. 5.16 приведены вспомогательные функции, определенные в заголовке `<algorithm>` и вычисляющие максимум и/или минимум из двух или более функций. Все функции `minmax()` и все функции для списков инициализации предусмотрены лишь стандартом C++11.

Таблица 5.16. Операции для вычисления минимума и максимума

Операция	Действие
`min(a,b)`	Возвращает минимум из *a* и *b*, сравнивая их с помощью оператора <
`min(a,b,cmp)`	Возвращает минимум из значений *a* и *b*, сравнивая их с помощью функции *cmp*
`min(initlist)`	Возвращает минимум из списка *initlist*, выполняя сравнение с помощью оператора <
`min(initlist,cmp)`	Возвращает минимум из списка *initlist*, выполняя сравнение с помощью функции *cmp*
`max(a,b)`	Возвращает максимум из значений *a* и *b*, выполняя сравнение с помощью оператора <
`max(a,b,cmp)`	Возвращает максимум из списка *initlist*, выполняя сравнение с помощью функции *cmp*
`max(initlist)`	Возвращает максимум из списка *initlist*, выполняя сравнение с помощью оператора <
`max(initlist,cmp)`	Возвращает максимум из списка *initlist*, выполняя сравнение с помощью функции *cmp*
`minmax(a,b)`	Возвращает максимум и минимум из значений *a* и *b*, сравнивая их с помощью оператора <
`minmax(a,b,cmp)`	Возвращает максимум и минимум из значений *a* и *b*, сравнивая их с помощью функции *cmp*
`minmax(initlist)`	Возвращает максимум и минимум из списка *initlist*, сравнивая их с помощью оператора <
`minmax(initlist,cmp)`	Возвращает максимум и минимум из списка *initlist*, сравнивая их с помощью функции *cmp*

Функция `minmax()` возвращает объект класса `pair<>` (см. раздел 5.1.1), в котором первым значением является минимум, а вторым — максимум. Для версий с двумя аргументами функции `min()` и `max()` возвращают первый элемент, если оба значения равны. Для списков инициализации функции `min()` и `max()` возвращают первый из нескольких минимальных или максимальных элементов. Функция `minmax()` возвращает пару, состоящую из чисел `a` и `b` для двух равных аргументов, и первый минимум, но последний максимум для списка инициализации. Однако хороший стиль программирования требует не полагаться на эти особенности функций.

Версии, получающие два значения, возвращают ссылку; версии, получающие список инициализации, возвращают копии значений.

```
namespace std {
   template <typename T>
      const T& min (const T& a, const T& b);
   template <typename T>
      T min (initializer_list<T> initlist);
   ...
}
```

Причина заключается в том, что в данном случае используется внутренняя временная переменная, поэтому возвращение ссылки привело бы к появлению висячего указателя.

Обе функции в качестве дополнительного аргумента принимают критерий сравнения.

```
namespace std {
   template <typename T, typename Compare>
      const T& min (const T& a, const T& b, Compare cmp);
   template <typename T, typename Compare>
      T min (initializer_list<T> initlist, Compare cmp);
   ...
}
```

Аргумент сравнения может быть функцией или функциональным объектом (см. раздел 6.10), сравнивающим два аргумента и возвращающим результат проверки того, является ли первый аргумент меньшим второго в определенном порядке.

Следующий пример демонстрирует использование функции для вычисления максимума, получающей в качестве аргумента специальную функцию для сравнения:

```
// util/minmax1.cpp

#include <algorithm>

// функция, сравнивающая два указателя путем сравнения
// значений, на которые они ссылаются
bool int_ptr_less (int* a, int* b)
{
   return *a < *b;
}

int main()
{
   int x = 17;
   int y = 42;
   int z = 33;
   int* px = &x;
   int* py = &y;
   int* pz = &z;

   // вызываем функцию max() со специальной функцией сравнения
   int* pmax = std::max (px, py, int_ptr_less);

   // вызываем функцию minmax() для списка инициализации
   // со специальной функцией сравнения
   std::pair<int*,int*> extremes = std::minmax ({px, py, pz},
                                                int_ptr_less);
   ...
}
```

В качестве альтернативы для задания критерия сравнения можно использовать новое языковое средство — лямбда-функцию, а с помощью ключевого слова `auto` избежать явного объявления типа возвращаемого значения.

```
auto extremes = std::minmax ({px, py, pz}, [](int*a, int*b) {
                                                return *a < *b;
                                });
```

Определения функций `min()` и `max()` требуют, чтобы оба типа совпадали. Таким образом, их нельзя вызывать для объектов, имеющих разные типы.

```
int i;
long l;
...
std::max(i,l);    // ОШИБКА: типы аргументов не совпадают
std::max({i,l});  // ОШИБКА: типы аргументов не совпадают
```

Однако можно явно уточнить тип с помощью шаблонных аргументов (задавая тем самым тип возвращаемого значения).

```
std::max<long>(i,l);   // OK
std::max<long>({i,l}); // OK
```

5.5.2. Обмен двух значений

Функция `swap()` обменивает два объекта. Общая реализации функции `swap()` определена в заголовке `<utility>` следующим образом[25]:

```
namespace std {
   template <typename T>
   inline void swap(T& a, T& b) ... {
      T tmp(std::move(a));
      a = std::move(b);
      b = std::move(tmp);
      }
}
```

Таким образом, если есть возможность, значения перемещаются или присваиваются путем перемещения (см. детали семантики перемещения в разделе 3.1.5). До появления стандарта C++11 значения присваивались или копировались всегда.

Используя эту функцию, можно обменять содержимое двух произвольных переменных `x` и `y`, выполнив вызов

```
std::swap(x,y);
```

Разумеется, этот вызов возможен только в том случае, если тип параметра допускает семантику перемещения или копирования.

Отметим, что функция `swap()` предоставляет спецификацию исключения (вот почему в предыдущих объявлениях использовался эллипсис `...`). Спецификация исключения функции `swap()` в общем случае имеет следующий вид[26]:

---

[25] До версии C++11 функция `swap()` была определена в заголовочном файле `<algorithm>`.

[26] Подробности, касающиеся ключевого слова `noexcept`, изложены в разделе 3.1.7, а свойства типов, использованные в данном примере, описаны в разделе 5.4.2.

```
noexcept(is_nothrow_move_constructible<T>::value &&
         is_nothrow_move_assignable<T>::value)
```

В соответствии со стандартом C++11 стандартная библиотека языка C++ предусматривает перегрузку для массивов.

```
namespace std {
   template <typename T, size_t N>
      void swap (T (&a)[N], T (&b)[N])
                noexcept(noexcept(swap(*a,*b)));
}
```

Большое преимущество функции `swap()` заключается в том, что она позволяет создавать специальные реализации для более сложных типов с помощью специализации шаблонов или перегрузки функций. Эти специальные реализации помогают сэкономить время, поменяв местами внутренние члены, а не присваивая объекты. Это касается, например, всех стандартных контейнеров (см. раздел 7.1.2) и строк (см. раздел 13.2.8). Например, реализация функции `swap()` для простого контейнера, содержащая только массив и количество его элементов, может выглядеть примерно так:

```
class MyContainer {
   private:
      int* elems;     // динамический массив элементов
      int numElems;  // количество элементов
   public:
      ...
      // реализация функции swap()
      void swap(MyContainer& x) {
         std::swap(elems,x.elems);
         std::swap(numElems,x.numElems);
      }
      ...
};

// перегруженная глобальная версия функции swap() для этого типа
inline void swap (MyContainer& c1, MyContainer& c2)
                  noexcept(noexcept(c1.swap(c2)))
{
   c1.swap(c2); // calls implementation of swap()
}
```

Итак, вызывая функцию `swap()` вместо непосредственного обмена значений, можно значительно повысить быстродействие программы. Если возникает возможность повысить производительность, всегда следует предусматривать специализацию функции `swap()` для своих типов.

Подчеркнем, что оба типа должны совпадать:

```
int i;
long l;
std::swap(i,l);      // ОШИБКА: типы аргументов не совпадают
int a1[10];
int a3[11];
std::swap(a1,a3);    // ОШИБКА: массивы имеют разные типы (разные размеры)
```

5.5.3. Вспомогательные операторы сравнения

Четыре шаблонных функции определяют операторы сравнения !=, >, <= и >= с помощью операторов == и <. Эти функции объявлены в заголовке <utility> и обычно определяются следующим образом:

```
namespace std {
   namespace rel_ops {
      template <typename T>
      inline bool operator!= (const T& x, const T& y) {
         return !(x == y);
      }
      template <typename T>
      inline bool operator> (const T& x, const T& y) {
         return y < x;
      }
      template <typename T>
      inline bool operator<= (const T& x, const T& y) {
         return !(y < x);
      }
      template <typename T>
      inline bool operator>= (const T& x, const T& y) {
         return !(x < y);
      }
   }
}
```

Для использования этих функций необходимо лишь определить операторы < и ==. Использование пространства имен std::rel_ops определяет другие операторы сравнения автоматически. Рассмотрим пример:

```
#include <utility>

class X {
   public:
      bool operator== (const X& x) const;
      bool operator< (const X& x) const;
     ...
};

void foo()
{
   using namespace std::rel_ops; // открывает доступ к операторам !=, > и др.
   X x1, x2;
   ...
   if (x1 != x2) { // OK
      ...
   }
   if (x1 > x2) { // OK
      ...
   }
}
```

Эти операторы определены в подпространстве имен `rel_ops` пространства имен `std`. Они находятся в отдельном пространстве имен, поэтому операции сравнения, определенные пользователем в глобальном пространстве имен, не создадут конфликта, даже если все идентификаторы в пространстве имен `std` станут глобальными после использования директивы

```
using namespace std; // операции не находятся в глобальной области видимости
```

С другой стороны, пользователи, желающие получить явный доступ к этим операциям, могут реализовать их, не используя правила неявного поиска.

```
using namespace std::rel_ops; // операции находятся
                              // в глобальной области видимости
```

Некоторые реализации определяют операции с двумя разными типами аргументов.

```
namespace std {
   namespace rel_ops {
      template <typename T1, typename T2>
      inline bool operator!=(const T1& x, const T2& y) {
         return !(x == y);
      }
   ...
   }
}
```

Преимущество такой реализации заключается в том, что в ней типы операндов могут быть разными. Достаточно лишь, чтобы их можно было сравнивать. Следует, однако, отметить, что такая реализация в стандартной библиотеке языка C++ не определена. Следовательно, такой код не будет переносимым.

5.6. Арифметика рациональных чисел на этапе компиляции

Класс `ratio<>`

В соответствии со стандартом C++11 стандартная библиотека языка C++ предусматривает интерфейс для дробей и арифметических операций над ними, выполняемых на этапе компиляции. Процитируем [N2661:Chrono] (с небольшими изменениями):[27]

Средство для работы с дробями является универсальным механизмом, предложенным Уолтером Е. Брауном, для простого и безопасного вычисления рациональных чисел на этапе компиляции. Класс ratio перехватывает все ошибки (например, деление на нуль и переполнение) на этапе компиляции. Он используется в библиотеках, связанных с измерениями продолжительности и времени [см. раздел 5.7], для эффективного

[27] Благодарю Уолтера Е. Брауна (Walter E. Brown), Говарда Хиннанта (Howard Hinnant), Джеффа Гарланда (Jeff Garland), и Марка Патерно (Marc Paterno) за любезное разрешение процитировать работу [*N2661:Chrono*] здесь и в следующем разделе, посвященном функциям работы со временем.

создания единиц времени. Его можно также применять в других "количественных" библиотеках (как стандартных, так и пользовательских) или в любом другом месте, где имеется рациональная константа, известная на этапе компиляции. Использование этого класса позволяет существенно снизить вероятность переполнения на этапе выполнения программы, поскольку дроби и результаты выполнения арифметических операций над ними всегда сокращаются до несократимой дроби.

Средство для работы с дробями описано в заголовке <ratio> в виде класса ratio<>, определенного следующим образом:

```
namespace std {
   template <intmax_t N, intmax_t D = 1>
   class ratio {
      public:
         static constexpr intmax_t num;
         static constexpr intmax_t den;

         typedef ratio<num,den> type;
   };
}
```

Тип intmax_t означает целочисленный тип со знаком, позволяющий представить любое целое число со знаком. Он определен в заголовке <cstdint> или <stdint.h> и имеет не менее 64 битов. Числитель и знаменатель являются открытыми и автоматически сокращаются до несократимой дроби. Например:

```
// util/ratio1.cpp

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

int main()
{
   typedef ratio<5,3> FiveThirds;
   cout << FiveThirds::num << "/" << FiveThirds::den << endl;

   typedef ratio<25,15> AlsoFiveThirds;
   cout << AlsoFiveThirds::num << "/" << AlsoFiveThirds::den << endl;

   ratio<42,42> one;
   cout << one.num << "/" << one.den << endl;

   ratio<0> zero;
   cout << zero.num << "/" << zero.den << endl;

   typedef ratio<7,-3> Neg;
   cout << Neg::num << "/" << Neg::den << endl;
}
```

Программа выводит следующие результаты:

```
5/3
5/3
1/1
0/1
-7/3
```

В табл. 5.17 перечислены операции над рациональными числами, которые можно выполнять на этапе компиляции. Четыре основных арифметических операции, +, -, * и /, определены как `ratio_add`, `ratio_subtract`, `ratio_multiply` и `ratio_divide`. Результат имеет тип `ratio<>`, поэтому статический член `type` выдает соответствующий тип. Например, результат следующего выражения равен `std::ratio<13,21>` (вычисленный как 6/21+7/21):

```
std::ratio_add<std::ratio<2,7>,std::ratio<2,6>>::type
```

Таблица 5.17. Операции класса `ratio<>`

Операция	Действие	Результат
`ratio_add`	Сокращенная сумма рациональных чисел	`ratio<>`
`ratio_subtract`	Сокращенная разность рациональных чисел	`ratio<>`
`ratio_multiply`	Сокращенное произведение рациональных чисел	`ratio<>`
`ratio_divide`	Сокращенное частное рациональных чисел	`ratio<>`
`ratio_equal`	Проверка ==	`true_type` или `false_type`
`ratio_not_equal`	Проверка !=	`true_type` или `false_type`
`ratio_less`	Проверка <	`true_type` или `false_type`
`ratio_less_equal`	Проверка <=	`true_type` или `false_type`
`ratio_greater`	Проверка >	`true_type` или `false_type`
`ratio_greater_equal`	Проверка >=	`true_type` или `false_type`

Кроме того, можно сравнивать два рациональных типа с помощью предикатов `ratio_equal`, `ratio_not_equal`, `ratio_less`, `ratio_less_equal`, `ratio_greater` или `ratio_greater_equal`. Как и свойства типов, тип результата порождается из типов `true_type` или `false_type` (см. раздел 5.4.2), поэтому их член `value` возвращает значение `true` или `false`.

```
ratio_equal<ratio<5,3>,ratio<25,15>>::value // возвращает значение true
```

Как было сказано выше, класс `ratio` перехватывает все ошибки, возникающие на этапе компиляции, такие как деление на нуль или переполнение. Например, выражение

```
ratio_multiply<ratio<1,numeric_limits<long long>::max()>,
               ratio<1,2>>::type
```

не скомпилируется, потому что умножение 1/*max* на 1/2 приводит к переполнению и результирующее значение знаменателя выходит за пределы, допустимые для данного типа.

Аналогично не будет скомпилировано и следующее значение, поскольку оно приводит к делению на нуль:

```
ratio_divide<fiveThirds,zero>::type
```

Отметим, однако, что следующее выражение будет скомпилировано, потому что неверное значение обнаруживается только при вычислении членов `type`, `num` или `den`:

```
ratio_divide<fiveThirds,zero>
```

Таблица 5.18. Предопределенные значения класса `ratio`

Имя	Значение
yocto	1/ 1 000 000 000 000 000 000 000 000 (не обязательно)
zepto	1/1000 000 000 000 000 000 000 (не обязательно)
atto	1/1000 000 000 000 000 000
femto	1/1000 000 000 000 000
pico	1/1000 000 000 000
nano	1/1000 000 000
micro	1/1000 000
milli	1/1000
centi	1/100
deci	1/10
deca	10
hecto	100
kilo	1 000
mega	1 000 000
giga	1 000 000 000
tera	1 000 000 000 000
peta	1 000 000 000 000 000
exa	1 000 000 000 000 000 000
zetta	1 000 000 000 000 000 000 000 (не обязательно)
yotta	1 000 000 000 000 000 000 000 000 (не обязательно)

Предопределенные значения рациональных чисел позволяют удобно задавать очень большие или очень маленькие числа (см. табл. 5.18). Они позволяют задавать большие числа без утомительного перечисления нулей. Например, выражение

```
std::nano
```

эквивалентно выражению

```
std::ratio<1,1000000000LL>
```

С помощью этого значения можно задавать, например, наносекунды (см. раздел 5.7.2). Единицы, помеченные как необязательные, определены, только если их допускает тип `intmax_t`.

5.7. Часы и таймеры

Одной из самых необходимых библиотек в любом языке программирования является библиотека для работы с датами и временем. Однако опыт показывает, что такую библиотеку труднее разработать, чем это кажется. Проблема заключается в степени гибкости и точности, которую должна обеспечить эта библиотека. В прошлом интерфейсы к системному времени, предусмотренные в языке C и в стандарте POSIX, переключались с секунд на миллисекунды, затем на микросекунды и наконец на наносекунды. Проблема заключалась в том, что при каждом таком переключении требовался новый интерфейс. По этой причине в стандарте C++11 была предложена нейтральная в смысле точности библиотека. Эта библиотека обычно называется *хронобиблиотекой* (библиотекой Chrono), потому что она определена в заголовочном файле `<chrono>`.

Кроме того, стандартная библиотека языка C++ содержит базовые интерфейсы языка C и стандарта POSIX для работы с календарным временем. В заключение отметим, что библиотека потоков, предоставляемая стандартом C++11, позволяет приостановить выполнение потока или программы (главного потока) на определенный период времени.

5.7.1. Обзор библиотеки Chrono

Библиотека Chrono разработана так, чтобы учесть тот факт, что таймеры и часы в разных системах разные и со временем могут становиться все более точными. Для того чтобы избежать введения нового типа для времени, скажем, через десять лет, как это произошло с библиотеками времени в стандарте POSIX, разработчики постарались реализовать концепцию нейтральной точности, отделив понятие интервала и момента времени ("timepoint") от конкретных часов. В результате ядро библиотеки Chrono состоит из следующих типов и понятий, служащих абстрактными механизмами для указания и обработки моментов и интервалов.

- *Интервал времени* (duration) определен как некоторое количество тактов определенной продолжительности (единицы времени). Например, интервал продолжительностью 3 минуты означает три такта по минуте. Другие примеры — "42 миллисекунды" или "86 400 секунд", представляющий интервал, равный одним суткам. Эта концепция позволяет также задавать такие периоды, как "1,5 раза по трети секунды", где 1,5 — количество тактов, а "треть секунды" — используемая единица времени.
- *Момент времени* (timepoint) определен как комбинация интервала и начала отсчета времени (так называемой эпохи). Типичным примером момента времени является "Полночь нового 2000-го года", описываемая как "1 262 300 400 секунд с 1 января 1970 года" (этот день является эпохой (epoch) для системных часов в системах UNIX и POSIX).
- Однако концепция момента времени параметризована *часами* (clock), представляющими собой объект, определяющий эпоху момента времени. Таким образом, разные часы имеют разные эпохи. Операторы, обрабатывающие несколько моментов времени, например, вычисляющие интервал между двумя моментами времени (их

разность), требуют использования одной и той же эпохи (часов). Кроме того, часы предоставляют удобную функцию для выдачи *текущего* момента времени.

Иначе говоря, момент времени определен как интервал времени, прошедшего после эпохи, заданной часами (см. рис. 5.4).

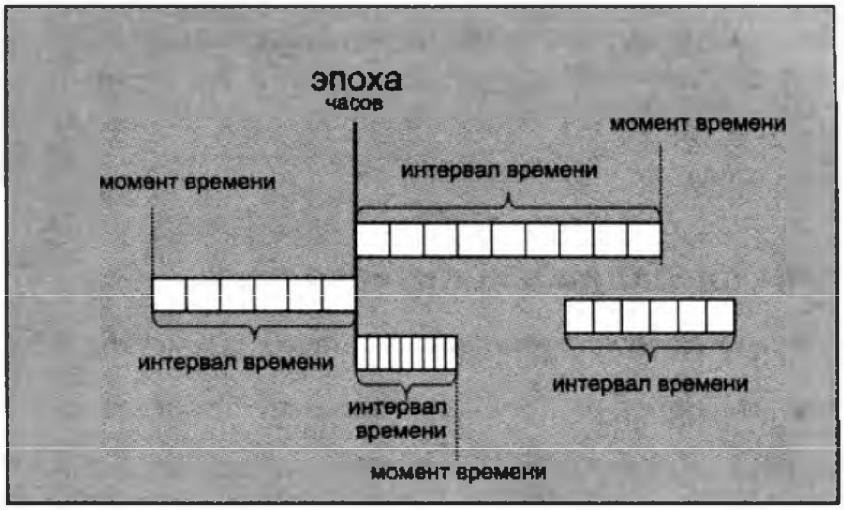


Рис. 5.4. Эпоха, интервалы и моменты времени

Более подробно о мотивации и разработке этих классов можно прочитать в работе [N2661:Chrono][28]. Рассмотрим подробнее эти типы и концепции.

Отметим, что все идентификаторы в библиотеке Chrono определены в пространстве имен `std::chrono`.

5.7.2. Интервалы времени

Интервал времени — это комбинация *значения*, представляющего количество тактов, и дроби, представляющей единицу времени в секундах. Для задания этой дроби используется класс `ratio` (см. раздел 5.6). Рассмотрим пример:

```
std::chrono::duration<int> twentySeconds(20);
std::chrono::duration<double,std::ratio<60>> halfAMinute(0.5);
std::chrono::duration<long,std::ratio<1,1000>> oneMillisecond(1);
```

где первый шаблонный аргумент определяет тип тактов, а необязательный второй шаблонный аргумент определяет тип единицы в секундах. Таким образом, первая строка использует в качестве типа единиц секунды, вторая — минуты ("60/1 секунд"), третья — миллисекунды ("1/1000 секунды").

Для удобства стандартная библиотека языка C++ содержит следующие определения типов:

```
namespace std {
   namespace chrono {
      typedef duration<signed int-type >= 64 битов,nano> nanoseconds;
      typedef duration<signed int-type >= 55 битов,micro> microseconds;
```

[28] Цитаты из работы [*N2661:Chrono*] приводятся с любезного разрешения авторов.

```
        typedef duration<signed int-type >= 45 битов,milli> milliseconds;
        typedef duration<signed int-type >= 35 битов> seconds;
        typedef duration<signed int-type >= 29 битов,ratio<60>> minutes;
        typedef duration<signed int-type >= 23 битов,ratio<3600>> hours;
    }
}
```

С их помощью легко задавать типичные периоды времени.

```
std::chrono::seconds twentySeconds(20);
std::chrono::hours aDay(24);
std::chrono::milliseconds oneMillisecond(1);
```

Арифметические операции над интервалами

Интервалы вычисляются вполне ожидаемым способом (см. табл. 5.19).

- Можно вычислить сумму, разность, произведение и частное двух интервалов.
- Можно складывать или вычитать такты и другие интервалы.
- Можно сравнивать два интервала.

Важным обстоятельством является то, что типы единицы для двух интервалов в таких операциях могут быть разными. Благодаря перегрузке класса `common_type<>` (см. раздел 5.4.1) для объектов класса `duration`, результирующий интервал будет измеряться в единицах измерения наибольшего общего знаменателя единиц обоих операндов. Например, после выполнения операторов

```
chrono::seconds d1(42);      // 42 секунды
chrono::milliseconds d2(10); // 10 миллисекунды
```

выражение

```
d1 - d2
```

дает интервал, длина которого равна 41 990 тактов в миллисекундах (1/1000 секунды).

В более общем случае после выполнения операторов

```
chrono::duration<int,ratio<1,3>> d1(1); // 1 такт по 1/3 секунды
chrono::duration<int,ratio<1,5>> d2(1); // 1 такт по 1/5 секунды
```

выражение

```
d1 + d2
```

создаст интервал, состоящий из 8 тактов по 1/15, а выражение

```
d1 < d2
```

будет иметь значение `false`. В обоих случаях интервал `d1` состоит из 5 тактов по 1/15 секунд, а `d2` — из 3 тактов по 1/15 секунд. Таким образом, сумма 3 и 5 равна 8, а 5 больше 3.

Таблица 5.19. Арифметические операции над объектами класса duration

Операция	Действие
`d1 + d2`	Вычисляет сумму двух интервалов `d1` и `d2`
`d1 - d2`	Вычисляет разность между интервалами `d1` и `d2`
`d * val`	Возвращает результат, равный произведению `val` на интервал `d`
`val * d`	Возвращает результат, равный произведению `val` на интервал `d`
`d / val`	Возвращает интервал `d`, деленный на значение `val`
`d1 / d2`	Вычисляет частное от деления интервала `d1` на интервал `d2`
`d % val`	Вычисляет остаток от деления интервала `d1` на значение `val`
`d % d2`	Вычисляет остаток от деления интервала `d` на интервал `d2`
`d1 == d2`	Возвращает результат проверки того, что интервал `d1` равен интервалу `d2`
`d1 != d2`	Возвращает результат проверки того, что интервал `d1` не равен интервалу `d2`
`d1 < d2`	Возвращает результат проверки того, что интервал `d1` короче интервала `d2`
`d1 <= d2`	Возвращает результат проверки того, что интервал `d1` не длиннее интервала `d2`
`d1 > d2`	Возвращает результат проверки того, что интервал `d1` длиннее интервала `d2`
`d1 <= d2`	Возвращает результат проверки того, что интервал `d1` не короче интервала `d2`
`++d`	Увеличивает интервал `d` на 1 такт
`d++`	Увеличивает интервал `d` на 1 такт
`--d`	Уменьшает интервал `d` на 1 такт
`d--`	Уменьшает интервал `d` на 1 такт
`d += d1`	Увеличивает интервал `d` на длину интервала `d1`
`d -= d1`	Уменьшает интервал `d` на длину интервала `d1`
`d *= val`	Умножает интервал `d` на значение `val`
`d /= val`	Делит интервал `d` на значение `val`
`d %= val`	Делит интервал `d` на значение `val` по модулю
`d %= d2`	Делит интервал `d` на интервал `d2` по модулю

Можно также преобразовать интервалы из одних единиц измерения в другие, поскольку в классе предусмотрено неявное преобразование типа. Таким образом, можно преобразовать часы в секунды, но не наоборот. Рассмотрим примеры:

```
std::chrono::seconds twentySeconds(20); // 20 секунд
std::chrono::hours   aDay(24);          // 24 часа
std::chrono::milliseconds ms(0);        // 0 миллисекунд
                                        // (без нуля значение не определено)
ms += twentySeconds + aDay;             // 86 420 000 миллисекунд
--ms;                                   // 86 419 999 миллисекунд
ms *= 2;                                // 172 839 998 миллисекунд
std::cout << ms.count() << " ms" << std::endl;
std::cout << std::chrono::nanoseconds(ms).count() << " ns" << std::endl;
```

Эти преобразования приводят к таким результатам:

```
172839998 ms
172839998000000 ns
```

Другие операции над интервалами

В предыдущем примере мы использовали функцию-член `count()` для вычисления текущего количества тактов — одну из многих операций над интервалами. В табл. 5.18 приведены все операции, члены и типы, доступные для интервалов, помимо арифметических операций, указанных в табл. 5.20. Отметим, что конструктор по умолчанию выполняет инициализацию интервала значением, предусмотренным по умолчанию (см. раздел 3.2.1), т.е. для элементарных типов начальное значение не определено.

Таблица 5.20. Другие операции и типы в классе `duration`

Операция	Действие
duration d	Конструктор по умолчанию; создает интервал (инициализированный по умолчанию)
duration d(*d2*)	Копирующий конструктор (интервал *d2* может иметь разные типы измерения)
duration d(*val*)	Создает интервал, состоящий из *val* тактов, имеющих тип, предусмотренный для интервала *d*
d = *d2*	Присваивает интервал *d2* интервалу *d* (возможно неявное преобразование)
d.`count()`	Возвращает количество тактов в интервале *d*
`duration_cast<`*D*`>`(*d*)	Возвращает интервал *d*, явно преобразованный в тип *D*
duration`::zero()`	Создает интервал нулевой длины
duration`::max()`	Создает максимально возможный интервал данного типа
duration`::min()`	Создает минимально возможный интервал данного типа
duration`::rep`	Возвращает тип тактов
duration`::period`	Возвращает тип единицы измерения

Эти члены можно использовать для определения удобной функции для оператора вывода << интервалов[29].

```
template <typename V, typename R>
ostream& operator << (ostream& s, const chrono::duration<V,R>& d)
{
   s << "[" << d.count() << " of " << R::num << "/"
                                   << R::den << "]";
   return s;
}
```

[29] Отметим, что этот оператор вывода не работает без механизма *ADL* (*argument-dependent lookup — поиск с учетом аргументов*) (см. раздел 15.11.1).

Здесь после вывода количества тактов с помощью функции count() мы выводим числитель и знаменатель используемой единицы измерения, которую класс ratio создает на этапе компиляции (см. раздел 5.6). Например, операторы

```
std::chrono::milliseconds d(42);
std::cout << d << std::endl;
```

выводят на экран строку

```
[42 of 1/1000]
```

Как видим, всегда существует возможность неявного преобразования в более точный тип единицы измерения. Однако неявное преобразование в менее точный тип единицы измерения невозможно, поскольку при этом происходит потеря информации. Например, при преобразовании целочисленного значения 42 010 миллисекунд в секунды результирующее целочисленное значение 42 означает потерю 10 миллисекунд. Однако такое преобразование можно выполнить принудительно с помощью оператора duration_cast. Например:

```
std::chrono::seconds sec(55);
std::chrono::minutes m1 = sec; // ОШИБКА
std::chrono::minutes m2 =
   std::chrono::duration_cast<std::chrono::minutes>(sec); // OK
```

В качестве другого примера отметим, что преобразование интервала с тактами, имеющими тип числа с плавающей точкой, также требует явного приведения при преобразовании в целочисленный интервал.

```
std::chrono::duration<double,std::ratio<60>> halfMin(0.5);
std::chrono::seconds s1 = halfMin; // ОШИБКА
std::chrono::seconds s2 =
   std::chrono::duration_cast<std::chrono::seconds>(halfMin); // OK
```

Типичный пример — код, разделяющий интервал на разные единицы. Например, следующий код разделяет интервал, состоящий из миллисекунд, на соответствующее количество часов, минут, секунд и миллисекунд (первая строка вывода начинается с выражения raw: и является результатом выполнения оператора вывода, которую мы только что определили).

```
using namespace std;
using namespace std::chrono;
milliseconds ms(7255042);

// разделяем интервал на часы, минуты, секунды и миллисекунды
hours hh = duration_cast<hours>(ms);
minutes mm = duration_cast<minutes>(ms % chrono::hours(1));
seconds ss = duration_cast<seconds>(ms % chrono::minutes(1));
milliseconds msec
           = duration_cast<milliseconds>(ms % chrono::seconds(1));

// выводим интервалы и значения:
cout << "raw: " << hh << "::" << mm << "::"
```

```
                << ss << "::" << msec << endl;
cout << " " << setfill('0') << setw(2) << hh.count() << "::"
                            << setw(2) << mm.count() << "::"
                            << setw(2) << ss.count() << "::"
                            << setw(3) << msec.count() << endl;
```

Здесь приведение

```
std::chrono::duration_cast<std::chrono::hours>(ms)
```

преобразовывает миллисекунды в часы, усекая, а не округляя значение. Благодаря оператору деления по модулю `%`, которому в качестве второго операнда можно передавать интервал, можно легко обработать оставшиеся миллисекунды с помощью оператора `ms % std::chrono::hours(1)`, которые затем преобразовываются в минуты. Таким образом, вывод этого кода выглядит следующим образом:

```
raw: [2 of 3600/1]::[0 of 60/1]::[55 of 1/1]::[42 of 1/1000]
     02::00::55::042
```

В заключение отметим, что класс `duration` содержит три статические функции: `zero()`, создающую интервал нулевой длины в секундах, а также `min()` и `max()`, создающие минимально и максимально возможные интервалы.

5.7.3. Часы и моменты времени

Отношения между моментами времени и часами довольно запутанные.

- *Часы* определяют эпоху и продолжительность такта. Например, часы могут иметь такт, измеряемый в миллисекундах, начиная с эпохи UNIX (1 января 1970 года), или в наносекундах, начиная с запуска программы. Кроме того, часы задают тип любого момента времени, определяемого по данным часам.
- Интерфейс часов содержит функцию `now()`, возвращающую объект для текущего момента времени.
- *Момент времени* представляет конкретный момент времени, связанный с положительным или отрицательным интервалом, измеренным по заданным часам. Таким образом, интервал "10 суток" и ассоциированные с ним часы с эпохой "1 января 1970 года" соответствует 11 января 1970 года.
- Интерфейс момента времени позволяет определить эпоху, минимальный и максимальный моменты времени, возможные для данных часов, а также арифметические операции над моментами времени.

Часы

В табл. 5.21 приведены определения типов и статические члены, требуемые для всех часов.

Таблица 5.21.Операции и типы часов

Операция	Действие
часы::duration	Возвращает тип интервала для часов
часы::rep	Возвращает тип такта (эквивалент *часы*::duration::rep)
часы::period	Возвращает тип единицы измерения (эквивалент *часы*::duration::period)
часы::time_point	Возвращает тип момента времени для часов
часы::is_steady	Возвращает значение true, если часы являются монотонными
часы::now()	Возвращает объект класса time_point для текущего момента времени

Стандартная библиотека языка C++ содержит три вида часов, имеющих собственный интерфейс.

1. Класс system_clock представляет моменты времени, связанные с обычными часами реального времени в текущей операционной системе. Эти часы также имеют вспомогательные функции to_time_t() и from_time_t(), чтобы конвертировать любой момент времени в объект типа time_t, предусмотренного в языке C, и наоборот. Это означает, что его можно конвертировать в календарную дату, и наоборот (см. раздел 5.7.4).
2. Класс steady_clock предоставляет гарантии того, что часы никогда не будут корректироваться[30]. Таким образом, значения моментов времени, измеряемых по этим часам, никогда не уменьшаются и возрастают равномерно по отношению к реальному времени.
3. Класс high_resolution_clock представляет часы с наименьшей продолжительностью такта, возможной в текущей системе.

Отметим, что стандарт не формулирует требований к точности, эпохе и диапазону (минимальному и максимальному моментам времени) перечисленных часов. Например, ваши системные часы могут иметь эпоху UNIX (1 января 1970 года), но это ничем не гарантируется. Если вам потребуется конкретная эпоха или моменты времени, которых нет у перечисленных часов, придется воспользоваться вспомогательными функциями.

Например, следующая функция выводит на экран свойства часов:

```
// util/clock.hpp

#include <chrono>
#include <iostream>
#include <iomanip>

template <typename C>
void printClockData ()
{
   using namespace std;

   cout << "- precision: ";
   // если момент времени меньше или равен одной миллисекунде
```

[30] Класс steady_clock сначала предполагалось назвать monotonic_clock.

```
   typedef typename C::period P; // Тип единицы времени
   if (ratio_less_equal<P,milli>::value) {
      // преобразовываем в миллисекунды и выводим на экран
      typedef typename ratio_multiply<P,kilo>::type TT;
      cout << fixed << double(TT::num)/TT::den
           << " milliseconds" << endl;
   }
   else {
   // выводим на экран в виде секунд
   cout << fixed << double(P::num)/P::den << " seconds" << endl;
   }
   cout << "- is_steady: " << boolalpha << C::is_steady << endl;
}
```

Эту функцию можно вызвать для разных часов, предусмотренных в стандартной библиотеке языка C++.

```
// util/clock1.cpp

#include <chrono>
#include "clock.hpp"

int main()


{
   std::cout << "system_clock: " << std::endl;
   printClockData<std::chrono::system_clock>();
   std::cout << "\nhigh_resolution_clock: " << std::endl;
   printClockData<std::chrono::high_resolution_clock>();
   std::cout << "\nsteady_clock: " << std::endl;
   printClockData<std::chrono::steady_clock>();
}
```

Эта программа может вывести, например, следующие строки:

```
system_clock:
- precision: 0.000100 milliseconds
- is_steady: false

high_resolution_clock:
- precision: 0.000100 milliseconds
- is_steady: true

steady_clock:
- precision: 1.000000 milliseconds
- is_steady: true
```

Здесь, например, системные часы или часы высокого разрешения имеют одинаковую точность, равную 100 наносекундам, в то время как монотонные часы используют миллисекунды. Кроме того, монотонные часы и часы высокого разрешения невозможно корректировать. Отметим, однако, что в разных системах могут быть совершенно разные ситуации. Например, часы высокого разрешения могут не отличаться от системных часов.

Класс steady_clock играет важную роль при сравнении разности между двумя моментами времени в программе, когда вы обрабатываете текущий момент времени. Например, после выполнения оператора

```
auto system_start = chrono::system_clock::now();
```

проверка, работает ли программа дольше одной минуты,

```
if (chrono::system_clock::now() > system_start + minutes(1))
```

может не сработать, потому что, если в это время происходит коррекция часов, оператор сравнения может вернуть значение false, даже если на самом деле программа работала дольше минуты. Аналогично вычисление времени работы программы

```
auto diff = chrono::system_clock::now() - system_start;
auto sec = chrono::duration_cast<chrono::seconds>(diff);
cout << "this program runs: " << s.count() << " seconds" << endl;
```

может привести к отрицательному интервалу времени, если за время проверки часы корректировались. По этой же причине использование таймеров, отличающихся от класса steady_clock, может привести к изменению интервала времени из-за коррекции системных часов (см. раздел 5.7.5).

Моменты времени

При работе с любыми часами — даже с часами, заданными пользователем, — можно обрабатывать моменты времени. Соответствующий интерфейс, параметризованный типом часов, предоставляет класс time_point.

```
namespace std {
   namespace chrono {
      template <typename Clock,
                typename Duration = typename Clock::duration>
                    class time_point;
   }
}
```

Особую роль играют конкретные моменты времени.

1. *Эпоха*, которую для каждых часов создает конструктор по умолчанию класса time_point.
2. *Текущее время*, которое для каждых часов выдает статическая функция-член now() (см. раздел 5.7.3).
3. *Минимальный момент времени*, который для каждых часов выдает статическая функция-член min() класса time_point.
4. *Максимальный момент времени*, который для каждый часов выдает статическая функция-член max() класса time_point.

Например, следующая программа присваивает эти моменты времени объекту tp и выводит на экран в календарных обозначениях:

```
// util/chrono1.cpp

#include <chrono>
#include <ctime>
#include <string>
#include <iostream>

std::string asString (const std::chrono::system_clock::time_point& tp)
{
   // преобразование в системное время:
   std::time_t t = std::chrono::system_clock::to_time_t(tp);
   std::string ts = std::ctime(&t); // преобразование в календарное время
   ts.resize(ts.size()-1);          // пропуск символа новой строки
   return ts;
}

int main()
{
   // вывод на экран эпохи для системных часов:
   std::chrono::system_clock::time_point tp;
   std::cout << "epoch: " << asString(tp) << std::endl;

  // вывод текущего времени:
   tp = std::chrono::system_clock::now();
   std::cout << "now: " << asString(tp) << std::endl;

  // вывод минимального времени для системных часов:
   tp = std::chrono::system_clock::time_point::min();
   std::cout << "min: " << asString(tp) << std::endl;

   // вывод максимального времени для системных часов:
   tp = std::chrono::system_clock::time_point::max();
   std::cout << "max: " << asString(tp) << std::endl;
}
```

После включения заголовочного файла <chrono> мы сначала объявляем вспомогательную функцию `asString()`, которая преобразовывает момент времени по системным часам в соответствующее календарное время. В коде

```
std::time_t t = std::chrono::system_clock::to_time_t(tp);
```

используется статическая вспомогательная функция `to_time_t()`, преобразовывающая момент времени в объект традиционного типа времени `time_t`, принятого в языке C и стандарте POSIX, который обычно представляет собой количество секунд, прошедших с эпохи UNIX, т.е. 1 января 1970 года (см. раздел 5.7.4). Затем оператор `std::string ts = std::ctime(&t);` с помощью функции `ctime()` преобразовывает его в календарное время, из которого инструкция `ts.resize(ts.size()-1);` удаляет пустую строку.

Отметим, что функция `ctime()` учитывает часовой пояс. Последствия этого факта мы вскоре обсудим. Кроме того, как правило, эта вспомогательная функция работает только с классом `system_clock`, единственным классом, имеющим интерфейс для преобразования в тип `time_t`, и наоборот. Для других часов такой интерфейс может работать, но не быть переносимым, поскольку другие часы не требуют использования эпохи системного времени в качестве внутренней эпохи.

Отметим также, что формат вывода моментов времени иногда целесообразно локализовать с помощью аспекта time_put. Детали и примеры описаны в разделе 16.4.3.

В функции main() тип объекта tp объявлен как

```
std::chrono::system_clock::time_point
```

что эквивалентно[31]

```
std::chrono::time_point<std::chrono::system_clock>
```

Таким образом, объект tp объявлен как момент времени по часам типа system_clock. Возможность задавать часы как шаблонный аргумент гарантирует, что арифметика моментов времени возможна только с одними и теми же часами (эпохой).

Эта программа может вывести следующий результат:

```
epoch: Thu Jan  1 01:00:00 1970
now:   Sun Jul 24 19:40:46 2011
min:   Sat Mar  5 18:27:38 1904
max:   Mon Oct 29 07:32:22 2035
```

Таким образом, конструктор по умолчанию, определяющий эпоху, создает момент времени, который функция asString() преобразовывает в строку

```
Thu Jan 1 01:00:00 1970
```

Отметим, что этот момент времени соответствует часу ночи, а не полуночи. Это может показаться удивительным, но вспомним, что преобразование в календарное время с помощью функции ctime() в функции asString() учитывает часовой пояс.

Таким образом, эпоха UNIX, использованная здесь, — которая, напомним, не всегда совпадает с эпохой системного времени, — началась в 00:00 по Гринвичу. В моем часовом поясе, в Германии, в этот момент был час ночи, поэтому здесь эпоха началась в час ночи 1 января 1970 года. Соответственно, если вы запустите эту программу у себя на компьютере, то можете получить другой вывод, потому что он зависит от вашего часового пояса, даже если ваша система использует ту же самую эпоху, что и системные часы.

Для работы с универсальным временем (UTC) вместо вызова функции ctime() (которая является всего лишь сокращением для вызова asctime(localtime(...)) (см. раздел 5.7.4)) следует использовать следующее преобразование:

```
std::string ts = std::asctime(gmtime(&t));
```

В данном случае вывод программы может оказаться таким:

```
epoch: Thu Jan  1 00:00:00 1970
now:   Sun Jul 24 17:40:46 2011
min:   Sat Mar  5 17:27:38 1904
max:   Mon Oct 29 06:32:22 2035
```

Да, здесь функция now() выдает разницу, равную двум часам, потому что этот момент времени соответствует летнему времени, которое в часовом поясе Германии создает двухчасовую разницу с универсальным временем.

---

[31] В соответствии со стандартом класс system_clock::time_point может быть также идентичным классу time_point<C2,system_clock::duration>, где C2 — другие часы, имеющие ту же самую эпоху, что и класс system_clock.

В принципе, объекты класса `time_point` должны иметь только один член, интервал, соответствующий эпохе связанных с ними часов. Этот момент времени можно запросить с помощью функции `time_since_epoch()`. Для выполнения арифметических операций над моментами времени предусмотрены полезные комбинации моментов времени и интервалов (табл. 5.22).

Таблица 5.22. Операции в классе `time_point`

Операция	Результат	Действие
момент_времени t	*момент времени*	Конструктор по умолчанию; создает момент времени, представляющий эпоху
момент_времени t(*tp2*)	момент времени	Создает момент времени, эквивалентный объекту *tp2* (единица измерения интервала может уменьшаться)
момент_времени t(*d*)	момент времени	Создает момент времени, соответствующий интервалу *d* после эпохи
`time_point_cast<C,D>(tp)`	момент времени	Преобразовывает объект *tp* в момент времени по часам *C*, соответствующий интервалу *D* (единица измерения интервала может увеличиваться)
tp += *d*	момент времени	Добавляет интервал *d* к текущему моменту времени *tp*
tp -= *d*	момент времени	Вычитает интервал *d* из текущего момента времени *tp*
tp + *d*	момент времени	Возвращает новый момент времени *tp* с добавленным интервалом *d*
d + *tp*	момент времени	Возвращает новый момент времени *tp* с добавленным интервалом *d*
tp - *d*	момент времени	Возвращает новый момент времени *tp* с вычтенным интервалом *d*
tp1 - *tp2*	момент времени	Возвращает интервал между моментами времени *tp1* и *tp2*
tp1 == *tp2*	`bool`	Возвращает результат проверки, равен ли момент времени *tp1* моменту времени *tp2*
tp1 != *tp2*	`bool`	Возвращает результат проверки, отличается ли момент времени *tp1* от момента времени *tp2*
tp1 < *tp2*	`bool`	Возвращает результат проверки, наступил ли момент времени *tp1* до момента времени *tp2*
tp1 <= *tp2*	`bool`	Возвращает результат проверки, не наступил ли момент времени *tp1* не позже момента времени *tp1*
tp1 > *tp2*	`bool`	Возвращает результат проверки, наступил ли момент времени *tp1* после момента времени *tp2*

Окончание табл. 5.22

Операция	Результат	Действие
tp1 >= *tp2*	bool	Возвращает результат проверки, не наступил ли момент времени *tp1* не ранее момента времени *tp1*
tp.time_since_epoch()	интервал	Возвращает интервал между эпохой и моментом времени *tp*
момент_времени::min()	момент времени	Возвращает первый возможный момент времени типа *момент_времени*
момент_времени::max()	момент времени	Возвращает последний возможный момент времени типа *момент_времени*

Несмотря на то что интерфейс использует класс ratio (см. раздел 5.6), гарантирующий, что переполнение при вычислении интервала сгенерирует ошибку на этапе компиляции, переполнение при работе с интервалами все же возможно.

Рассмотрим следующий пример:

```
// util/chrono2.cpp

#include <chrono>
#include <ctime>
#include <iostream>
#include <string>
using namespace std;

string asString (const chrono::system_clock::time_point& tp)
{
   time_t t = chrono::system_clock::to_time_t(tp); // преобразуем
                                                   // в системное время
   string ts = ctime(&t);          // преобразуем в календарное время
   ts.resize(ts.size()-1);         // удаляем завершающий символ новой строки
   return ts;
}

int main()
{
   // определяем тип интервала для представления дней:
   typedef chrono::duration<int,ratio<3600*24>> Days;

   // определяем эпоху системных часов
   chrono::time_point<chrono::system_clock> tp;
   cout << "epoch: " << asString(tp) << endl;

   // добавляем одни сутки, 23 часа и 55 минут
   tp += Days(1) + chrono::hours(23) + chrono::minutes(55);
   cout << "later: " << asString(tp) << endl;

   // вычисляем длительность до эпохи в минутах и сутках:
   auto diff = tp - chrono::system_clock::time_point();
   cout << "diff: "
        << chrono::duration_cast<chrono::minutes>(diff).count()
        << " minute(s)" << endl;
   Days days = chrono::duration_cast<Days>(diff);
```

```
    cout << "diff: " << days.count() << " day(s)" << endl;

    // вычитаем один год (надеясь, что результат будет корректным
    // и год не високосный)
    tp -= chrono::hours(24*365);
    cout << "-1 year: " << asString(tp) << endl;

    // вычитаем 50 лет (надеясь, что результат будет корректным
    // и игнорируя високосные годы)
    tp -= chrono::duration<int,ratio<3600*24*365>>(50);
    cout << "-50 years: " << asString(tp) << endl;

    // вычитаем 50 лет (надеясь, что результат будет корректным
    // и игнорируя високосные годы)
    tp -= chrono::duration<int,ratio<3600*24*365>>(50);
    cout << "-50 years: " << asString(tp) << endl;
    }
```

Во-первых, выражения, такие как

```
tp = tp + Days(1) + chrono::hours(23) + chrono::minutes(55);
```

или

```
tp -= chrono::hours(24*365);
```

позволяют корректировать моменты времени с помощью арифметических операций над моментами времени.

Поскольку точность системных часов обычно выше, чем минуты и сутки, необходимо явно привести разницу между двумя моментами времени к суткам.

```
auto diff = tp - chrono::system_clock::time_point();
Days days = chrono::duration_cast<Days>(diff);
```

Однако следует подчеркнуть, что эти операции не проверяют возможность переполнения. На нашем компьютере результат работы программы был следующим:

```
epoch:     Thu Jan  1 01:00:00 1970
later:     Sat Jan  3 00:55:00 1970
diff:      2875 minute(s)
diff:      1 day(s)
-1 year:   Fri Jan  3 00:55:00 1969
-50 years: Thu Jan 16 00:55:00 1919
-50 years: Sat Mar  5 07:23:16 2005
```

Итак, можно сделать соответствующие выводы.

- Приведение к требуемой единице измерения, которая обычно имеет целочисленный тип, использует оператор `static_cast<>`, а это означает округление. По этой причине интервал, состоящий из 47 часов и 55 минут, после преобразования оказывается равным одним суткам.
- Вычитание 50 лет, состоящих из 365 суток, не учитывает високосные годы, поэтому результат равен 16 января, а не 3 января.

- При повторном вычитании 50 лет момент времени оказывается меньше минимального, который на нашем компьютере соответствует 5 марта 1904 года (см. раздел 5.7.3), поэтому результат равен 2005. В данном случае никаких ошибок нет.

Этот пример демонстрирует, что библиотека Chrono является библиотекой интервалов и моментов времени, а не дат и времени. Вы можете вычислять интервалы и моменты времени, но должны учитывать эпоху, минимальный и максимальный моменты времени, високосные годы и секунды координации.

5.7.4. Функции для работы с датами и временем в языке C и стандарте POSIX

В стандартной библиотеке языка C++ есть также стандартные интерфейсы C и POSIX для работы с датами и временем. Макросы и типы из заголовка `<ctime>`, а также функции из заголовочного файла `<time.h>` доступны в пространстве имен `std`. Типы и функции для работы с датами и временем перечислены в табл. 5.23. Кроме того, макрос `CLOCKS_PER_SEC` определяет тип единицы измерения для функции `clock()` (которая возвращает прошедшее время работы центрального процессора, измеренное в `1/CLOCKS_PER_SEC` долях секунд). Подробности и примеры, касающиеся функций и типов для работы с датами и временем, описаны в разделе 16.4.3.

Таблица 5.23. Определения в заголовочном файле `<ctime>`

Идентификатор	Смысл
`clock_t`	Тип числовых величин для измерения прошедшего времени работы центрального процессора, возвращаемых функцией `clock()`
`time_t`	Тип числовых величин, представляющих моменты времени
`struct tm`	Тип сокращенного календарного времени
`clock()`	Возвращает прошедшее время работы центрального процессора, измеренное в `1/CLOCKS_PER_SEC` долях секунды
`time()`	Возвращает текущее время как числовую величину
`difftime()`	Возвращает разницу между двумя моментами времени типа `time_t`, измеренную в секундах и представленную как число типа `double`
`localtime()`	Преобразовывает объект типа `time_t` в `struct tm` с учетом часового пояса
`gmtime()`	Преобразовывает объект типа `time_t` в `struct tm` без учета часового пояса
`asctime()`	Преобразовывает объект типа `struct tm` в стандартную строку, представляющую календарное время
`strftime()`	Преобразовывает объект типа `struct tm` в строку, представляющую календарное время и определенную пользователем
`ctime()`	Преобразовывает объект типа `time_t` в стандартную строку, пðåäñòàâëÿþùóþ календарное время с учетом часового пояса (сокращение для вызова функции `asctime(localtime(t))`)
`mktime()`	Преобразовывает объект типа `struct tm` в объект типа `time_t` и вычисляет день недели и год

Отметим, что тип time_t обычно выражает количество секунд, прошедших с эпохи UNIX, т.е. с 1 января 1970 года. Однако в соответствии со стандартами языков C и C++ это свойство не гарантируется.

Преобразования между моментами времени и календарным временем

Вспомогательная функция для преобразования момента времени в строку, представляющую календарное время, уже обсуждалась в разделе 5.7.3. Рассмотрим заголовочный файл, который также позволяет конвертировать календарное время в моменты времени.

```
// util/timepoint.hpp

#include <chrono>
#include <ctime>
#include <string>

// преобразование момента времени по системным часам
//в строку календарного времени
inline
std::string asString (const std::chrono::system_clock::time_point& tp)
{
   // преобразование в системное время:
   std::time_t t = std::chrono::system_clock::to_time_t(tp);
   std::string ts = ctime(&t); // преобразование в календарное время
   ts.resize(ts.size()-1);     // опускаем завершающий символ новой строки
   return ts;
}

// преобразование календарного времени
// в момент времени по системным часам
inline
std::chrono::system_clock::time_point
makeTimePoint (int year, int mon, int day,
int hour, int min, int sec=0)
{
   struct std::tm t;
   t.tm_sec = sec;         // секунд минуты (0 .. 59 и
                           // 60 для секунд координации)
   t.tm_min = min;         // минута часа (0 .. 59)
   t.tm_hour = hour;       // час суток (0 .. 23)
   t.tm_mday = day;        // день месяца (1 .. 31)
   t.tm_mon = mon-1;       // месяц года (0 .. 11)
   t.tm_year = year-1900; // год, считая от 1900
   t.tm_isdst = -1;        // определить, учитывается ли летнее время
   std::time_t tt = std::mktime(&t);
   if (tt == -1) {
      throw "no valid system time";
   }
   return std::chrono::system_clock::from_time_t(tt);
}
```

Эти функции иллюстрируются следующей программой:

```
// util/timepoint1.cpp
#include <chrono>
#include <iostream>
#include "timepoint.hpp"

int main()
{
   auto tp1 = makeTimePoint(2010,01,01,00,00);
   std::cout << asString(tp1) << std::endl;
   auto tp2 = makeTimePoint(2011,05,23,13,44);
   std::cout << asString(tp2) << std::endl;
}
```

Эта программа выдает следующие результаты:

```
Fri Jan  1 00:00:00 2010
Mon May 23 13:44:00 2011
```

Еще раз отметим, что функции `makeTimePoint()` и `asString()` учитывают часовой пояс. По этой причине дата, передаваемая функции `makeTimePoint()`, совпадает с результатом работы функции `asString()`. Кроме того, летнее время никакой роли не играет (передача отрицательного значения `t.tm_isdst` в функцию `makeTimePoint()` вынуждает функцию `mktime()` самостоятельно определить, учитывается ли летнее время при измерениях).

Для того чтобы функция `asString()` использовала не календарное, а универсальное время, следует использовать функцию `asctime(gmtime(...))`, а не `ctime(...)`. Для функции `mktime()` не существует механизма, заставляющего ее использовать время UTC, поэтому функция `makeTimePoint()` всегда учитывает часовой пояс.

Использование локальных установок для интернационализации чтения и записи показаний времени описано в разделе 16.4.3.

5.7.5. Блокировка с помощью таймеров

Интервалы и моменты времени можно использовать для блокировки потоков или программы (т.е. главного потока). Эти блоки могут быть безусловными, а могут использоваться для указания максимального интервала ожидания блокировки, переменной условия или завершения другого потока (см. главу 18).

- Функции `sleep_for()` и `sleep_until()` являются членами класса `this_thread` и предназначены для блокировки потоков (см. раздел 18.3.7).
- Функции `try_lock_for()` и `try_lock_until()` предназначены для задания максимального интервала ожидания для мьютекса (см. раздел 18.5.1).
- Функции `wait_for()` и `wait_until()` предназначены для задания максимального интервала при ожидании переменной условия или будущего момента времени (см. разделы 18.1.1 и 18.6.4).

Все блокирующие функции, имена которых завершаются суффиксом `_for()`, используют как аргумент интервал, а функции, имена которых завершаются суффиксом `_until()`, получают в качестве аргумента момент времени. Например, оператор

```
this_thread::sleep_for(chrono::seconds(10));
```

блокирует текущий поток, который может быть главным, на 10 секунд, а оператор

```
this_thread::sleep_until(chrono::system_clock::now()
+ chrono::seconds(10));
```

блокирует текущий поток, пока системные часы не достигнут отметки 10 секунд, считая от текущего момента.

Несмотря на то что все эти вызовы могут казаться одинаковыми, это не так! Для всех функций с суффиксом `_until()` при передаче аргумента допускается корректировка времени. Если на протяжении 10 секунд после вызова функции `sleep_until()` системные часы будут скорректированы, то задержка также будет уточнена соответствующим образом. Если, например, перевести системные часы назад на один час, то программа будет заблокирована на 60 минут и 10 секунд. Если же перевести часы вперед больше чем на 10 секунд, то таймер немедленно остановится.

При использовании функции с суффиксом `_for()`, например `sleep_for()`, получающей интервал, а также при использовании класса `steady_clock` корректировка системных часов, *как правило*, не влияет на продолжительность работы таймера. Однако если аппаратное обеспечение не имеет монотонных часов и поэтому нет возможности подсчитывать секунды независимо от потенциально скорректированного системного времени, то корректировки времени влияют также и на функции с суффиксом `_for()`.

Все таймеры могут быть неточными. Любой таймер может отставать, потому что система периодически проверяет просроченные таймеры, обрабатывает их и делает прерывания. Таким образом, интервалы таймеров могут равняться заданному периоду плюс периоду, который зависит от качества их реализации и текущей ситуации.

5.8. Заголовочные файлы <cstddef>, <cstdlib> и <cstring>

Следующие заголовочные файлы, совместимые с языком C, часто используются в программах на языке C++: `<cstddef>`, `<cstdlib>` и `<cstring>`. Они представляют собой версии заголовочных файлов `<stddef.h>`, `<stdlib.h>` и `<string.h>` из языка C, предназначенные для языка C++. В них определены некоторые общие константы, макросы, типы и функции.

5.8.1. Определения в заголовочном файле <cstddef>

В табл. 5.24 показаны определения из заголовочного файла `<cstddef>`. До появления стандарта C++11 значение `NULL` часто использовалось для описания ситуации, в которой указатель ни на что не ссылался. В стандарте C++11 эта семантика описывается значением `nullptr` (см. раздел 3.1.1).

Обратите внимание на то, что значение `NULL` в языке C++ гарантированно равно `0` (независимо от типа: `int` или `long`). В языке C значение `NULL` часто определяется как `(void*)0`. Однако в языке C++ это выражение является некорректным, потому что требуется, чтобы тип значения `NULL` был целочисленным. В противном случае значение `NULL` нельзя присвоить указателю. Это происходит потому, что в языке C++ нет автоматического преобразования типа `void*` в любой другой тип. В стандарте C++11 вместо `NULL`

следует использовать nullptr (см. раздел 3.1.1)[32]. Отметим также, что значение NULL также определено в заголовочных файлах <cstdio>, <cstdlib>, <cstring>, <ctime>, <cwchar> и <clocale>.

Таблица 5.24. Определения в заголовочном файле <cstddef>

Идентификатор	Смысл
NULL	Значение пустого указателя
nullptr_t	Тип значения nullptr (по стандарту C++11)
size_t	Беззнаковый тип для единиц размера, например для количества элементов
ptrdiff_t	Знаковый тип для разности между указателями
max_align_t	Тип для максимального выравнивания во всех контекстах (по стандарту C++11)
offsetof(*type*, *mem*)	Смещение члена *mem* в структуре или объединении *type*

5.8.2. Определения в заголовочном файле <cstdlib>

В табл. 5.25 приведены наиболее важные определения в заголовочном файле <cstdlib>. Две константы, EXIT_SUCCESS и EXIT_FAILURE, определены как аргументы функции exit() и могут использоваться в качестве значения, возвращаемого функцией main().

Таблица 5.25. Определения в заголовочном файле <cstdlib>

Определение	Смысл
EXIT_SUCCESS	Нормальное завершение программы
EXIT_FAILURE	Ненормальное завершение программы
exit (int *status*)	Выход из программы (очистка статических объектов)
quick_exit (int *status*)	Выход из программы с очисткой, предусмотренной функцией at_quick_exit() (по стандарту C++11)
_Exit (int *status*)	Выход из программы без очистки (по стандарту C++11)
abort()	Аварийное завершение программы (может привести к краху в некоторых системах)
atexit (void (**func*)())	Вызов функции *func* при выходе из программы
at_quick_exit (void (**func*)())	Вызов функции *func* при выполнении функции quick_exit() (по стандарту C++11)

При нормальном завершении программы функции, зарегистрированные функцией atexit(), вызываются в порядке, обратном порядку их регистрации. При этом не имеет значения, завершается ли программа функцией exit() или достигнут конец функции main(). Аргументы функциям не передаются.

[32] Из-за путаницы с типом значения NULL некоторые люди и руководства по стилю программирования рекомендуют вообще не использовать значение NULL в программах на языке C++. Вместо него лучше использовать 0 или специальную константу, определенную пользователем, например NIL. К счастью, концепция nullptr решила эту проблему.

Функции `exit()` и `abort()` прерывают выполнение программы в любой функции без возвращения в функцию `main()`.

- Функция `exit()` уничтожает все статические объекты, очищает все буфера, закрывает все каналы ввода-вывода и завершает выполнение программы, включая вызовы функций `atexit()`. Если функции, переданные функции `atexit()`, генерируют исключения, вызывается функция `terminate()`.
- Функция `abort()` немедленно прекращает выполнение программы без очистки.

Ни одна из этих функций не удаляет локальные объекты, потому что при их вызове не происходит раскручивание стека. Для того чтобы гарантированно вызывались деструкторы всех локальных объектов, необходимо использовать исключения или обычный механизм возврата управления и выхода из функции `main()`.

В стандарте C++11 семантика функции `quick_exit()` предусматривает не удаление объектов, а вызовы функций, зарегистрированных для вызова в функции `at_quick_exit()`, в порядке, обратном порядку их регистрации, и вызов функции `_Exit()`, прекращающей работу программы без удаления объектов и очистки[33]. Это означает, что функции `quick_exit()` и `_Exit()` не очищают стандартные файловые буфера (стандартного вывода и ошибок).

В языке C++ обычным способом аварийного прекращения работы программы, представляющего собой неожиданный выход в противоположность ожидаемому выходу с выдачей сигналов об ошибках, является вызов функции `std::terminate()`, которая по умолчанию вызывается функцией `abort()`. Это происходит, например, если деструктор или функция, объявленная с ключевым словом `noexcept` (см. раздел 3.1.7), генерирует исключение.

5.8.3. Определения в заголовочном файле <cstring>

В табл. 5.26 приведены наиболее важные определения заголовочного файла `<cstring>`: низкоуровневые функции для установки, копирования и перемещения участков памяти. Одним из применений этих функций являются свойства символов (см. раздел 16.1.4).

Таблица 5.26. Определения в заголовочном файле <cstring>

Определение	Смысл
`memchr(const void* ptr, int c, size_t len)`	Находит символ *c* среди первых *len* байтов, на которые ссылается указатель *ptr*
`memcmp(const void* ptr1, const void* ptr2, size_t len)`	Сравнивает *len* байтов памяти, на которую ссылаются указатели *ptr1* и *ptr2*
`memcpy(void* toPtr, const void* fromPtr, size_t len)`	Копирует *len* байтов из памяти, на которую ссылается указатель *fromPtr*, в память, на которую ссылается указатель *Ptr*
`memmove(void* toPtr, const void* fromPtr, size_t len)`	Копирует *len* байтов из памяти, на которую ссылается указатель *fromPtr*, в память, на которую ссылается указатель *toPtr* (области могут перекрываться)
`memset(void* ptr, int c, size_t len)`	Присваивает символ *c* первым *len* байтам памяти, на которую ссылается указатель *ptr*

[33] Этот механизм позволяет предотвратить доступ отсоединенных потоков к глобальным/статическим объектам (см. раздел 18.2.1).

Глава 6

Стандартная библиотека шаблонов

Стандартная библиотека шаблонов (standard template library — STL) является ядром стандартной библиотеки языка C++, повлиявшим на всю ее архитектуру. STL — это библиотека универсальных компонентов для управления коллекциями данных с помощью современных и эффективных алгоритмов. Она позволяет программистам использовать самые современные достижения в области структур данных и алгоритмов, не вникая в то, как они работают.

С точки зрения программиста библиотека STL содержит совокупность классов коллекций для различных целей и набор алгоритмов для работы с ними. Все компоненты STL являются шаблонами, поэтому их можно использовать для произвольных типов элементов. Однако это не все. Библиотека STL образует основу для создания других классов коллекций и алгоритмов, работающих совместно с существующими классами коллекций и алгоритмами. Благодаря этому библиотека STL поднимает язык C++ на новый уровень абстракции. Забудьте о программировании динамических массивов, связанных списков, бинарных деревьев и хеш-таблиц; забудьте о программировании разных алгоритмов поиска. Для использования требуемой коллекции достаточно определить соответствующий контейнер и вызвать соответствующие функции-члены и алгоритм для обработки данных.

Однако за гибкость библиотеки STL приходится расплачиваться ее сложностью. По этой причине описание библиотеки STL занимает в этой книге несколько глав. В настоящей главе излагается общая концепция библиотеки STL и объясняются приемы программирования, необходимые для работы с ней. Первые примеры демонстрируют использование библиотеки STL и главные моменты, на которые следует обратить внимание. В главах 7–11 подробно обсуждаются компоненты библиотеки STL (контейнеры, итераторы, функциональные объекты и алгоритмы) и приводятся дополнительные примеры.

6.1. Компоненты библиотеки STL

В основе библиотеки STL лежит взаимодействие разных хорошо структурированных компонентов, главными среди которых являются контейнеры, итераторы и алгоритмы.

- **Контейнеры** используются для управления коллекциями объектов определенного типа. Каждый вид контейнеров имеет свои достоинства и недостатки, поэтому наличие разных контейнеров отражает разные требования к коллекциям, существующие в программах. Контейнеры могут быть реализованы как массивы или связанные списки или могут иметь специальный ключ для каждого элемента.
- **Итераторы** используются для обхода элементов в коллекциях объектов. Этими коллекциями могут быть контейнеры или подмножества контейнеров. Основное преимущество итераторов заключается в том, что они предоставляют небольшой и в то

же время универсальный интерфейс для произвольного типа контейнера. Например, одна из операций этого интерфейса перемещает итератор на следующий элемент в коллекции. Выполнение этой операции не зависит от внутренней структуры коллекции. Чем бы ни была коллекция — массивом, деревом или хеш-таблицей, — эта операция работает одинаково, поскольку каждый контейнер определяет свой собственный тип итератора, который просто "правильно работает", поскольку знает внутреннюю структуру своего контейнера.

Интерфейс итераторов почти совпадает с интерфейсом обычных указателей. Для перехода итератора к следующему элементу выполняется операция ++, а для доступа к значению, на которое ссылается итератор, — операция *. Итак, итератор можно рассматривать как разновидность интеллектуального указателя, преобразующего команду "перейти к следующему элементу" в подходящую операцию.

- **Алгоритмы** предназначены для обработки элементов коллекций. Например, алгоритмы могут искать, сортировать, модифицировать и просто использовать элементы для разных целей. Алгоритмы используют итераторы. Таким образом, поскольку интерфейс итераторов является общим для всех типов контейнеров, алгоритм достаточно написать один раз, и он будет работать с любым контейнером

  Для того чтобы повысить гибкость алгоритмов, их можно использовать в сочетании со вспомогательными функциями, которые вызываются алгоритмами. Таким образом, универсальный алгоритм можно использовать для решения своей задачи, даже если эта задача является очень специфичной или сложной. Например, программист может задать особый критерий поиска или особую операцию для объединения элементов. С появлением стандарта C++11, который включает лямбда-функции, появилась возможность описывать практически любую функциональность при обходе элементов контейнера.

Концепция библиотеки STL основана на разделении данных и операций. Данные управляются контейнерными классами, а операции определяются настраиваемыми алгоритмами. Связующим звеном между этими двумя компонентами являются итераторы. Они позволяют алгоритму взаимодействовать с любым контейнером (рис. 6.1).

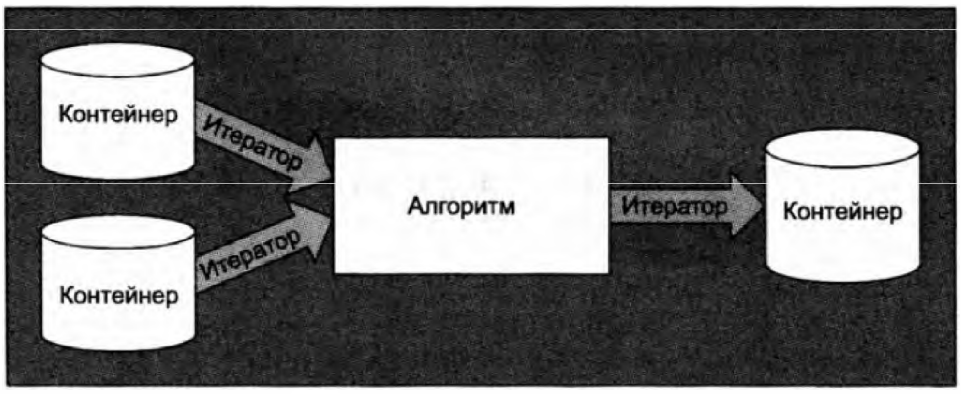


Рис. 6.1. Компоненты библиотеки STL

В некотором смысле концепция библиотеки STL противоречит исходной идее объектно-ориентированного программирования: STL *отделяет* данные от алгоритмов, а не объединяет их. Однако для этого есть весомая причина. В принципе, программист может

объединять любой контейнер с любым алгоритмом, так что результат будет очень гибким и в то же время компактным.

Одной из основных особенностей библиотеки STL является то, что все компоненты работают с произвольными типами. Как следует из названия "стандартная библиотека шаблонов", все компоненты являются шаблонами, допускающими использование любого типа, при условии, что этот тип способен выполнять требуемые операции. Таким образом, библиотека STL — яркий пример концепции *обобщенного программирования* (generic programming). Контейнеры и алгоритмы являются обобщенными по отношению к произвольным типам и классам соответственно.

В библиотеке STL есть еще более универсальные компоненты. С помощью некоторых *адаптеров* и *функциональных объектов* (или *функторов*) программист может расширять, ограничивать или настраивать алгоритмы и интерфейсы для конкретных целей. Впрочем, мы немного забежали вперед. Сначала рассмотрим эту концепцию на примерах. Вероятно, это лучший способ понять, как работает библиотека STL, и освоить ее.

6.2. Контейнеры

Контейнерные классы, или *контейнеры*, управляют коллекциями элементов. Для разных целей в библиотеке STL предусмотрены разные контейнеры, как показано на рис. 6.2.

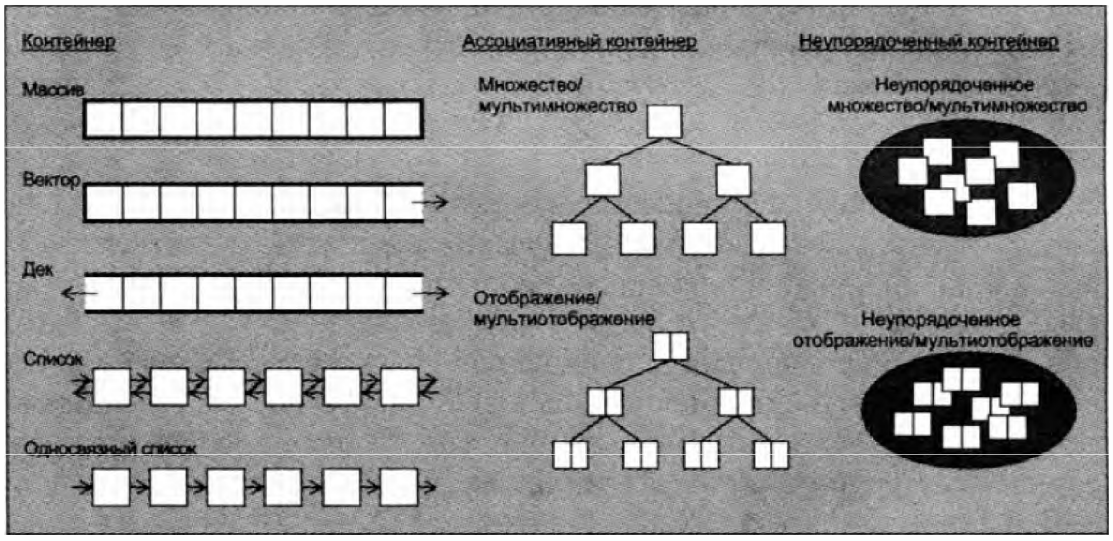


Рис. 6.2. Контейнерные типы в библиотеке STL

Существуют три разновидности контейнеров.

1. **Последовательные контейнеры** — это *упорядоченные коллекции*, в которых каждый элемент занимает определенную позицию. Эта позиция зависит от времени и места вставки, но не зависит от значения элемента. Например, если вставить шесть элементов в упорядоченную коллекцию, добавляя каждый элемент в конец коллекции, то эти элементы будут следовать в точном порядке их вставки. Библиотека STL содержит пять стандартных контейнерных классов: `array`, `vector`, `deque`, `list` и `forward_list`.[1]

[1] Класс `array` был введен в документе TR1, а класс `forward_list` — в стандарте C++11.

2. **Ассоциативные контейнеры** — это *упорядоченные коллекции*, в которых позиция элемента зависит от его значения (или ключа, если элемент представляет собой пару "ключ–значение") в соответствии с определенным критерием сортировки. Если вставить шесть элементов в такую коллекцию, то порядок их следования будет определен их значениями. В этом случае порядок вставки элементов не имеет значения. Библиотека STL содержит четыре стандартных ассоциативных контейнерных класса: `set`, `multiset`, `map` и `multimap`.
3. **Неупорядоченные (ассоциативные) контейнеры** — это *неупорядоченные коллекции*, в которых позиция элемента не имеет значения. В этом случае смысл имеет только один вопрос: принадлежит ли конкретный элемент такой коллекции. Ни порядок вставки, ни значение вставленного элемента не влияет на его позицию. Его позиция со временем может изменяться. Таким образом, если вставить в такую коллекцию шесть элементов, то их порядок будет неопределенным и со временем может измениться. Библиотека STL содержит четыре стандартных неупорядоченных контейнерных классов: `unordered_set`, `unordered_multiset`, `unordered_map` и `unordered_multimap`.

Неупорядоченные контейнеры были введены в документе TR1 и вызвали небольшую путаницу в контейнерной терминологии. Официально неупорядоченные контейнеры относятся к категории "неупорядоченные ассоциативные контейнеры". Не совсем ясно, что в этом названии подразумевается под ассоциативным контейнером: термин, объединяющий упорядоченные и неупорядоченные ассоциативные контейнеры, или аналог неупорядоченных контейнеров? Ответ часто зависит от контекста. В книге под ассоциативным контейнером подразумевается упорядоченный ассоциативный контейнер (в старом смысле), а термин "неупорядоченные контейнеры" употребляют без промежуточного слова "ассоциативные".

Три категории контейнеров, введенные выше, представляют собой логические группы, зависящие от определения порядка следования элементов. В соответствии с такой точкой зрения ассоциативный контейнер можно считать особой разновидностью последовательного контейнера, потому что упорядоченные коллекции имеют дополнительную возможность устанавливать порядок следования элементов в соответствии со специальным критерием сортировки. Это вполне естественно, особенно при использовании других библиотек классов коллекций, например, библиотеки языка Smalltalk или библиотеки NIHCL[2], в которых отсортированные коллекции являются производными от упорядоченных коллекций. Однако типы коллекций в библиотеке STL резко отличаются друг от друга и имеют совершенно разные реализации, не являющиеся производными друг от друга.

- Последовательные контейнеры обычно реализуются как массивы или связанные списки.
- Ассоциативные контейнеры, как правило, реализуются как бинарные деревья.
- Неупорядоченные контейнеры обычно реализуются как хеш-таблицы.

Строго говоря, в стандартной библиотеке C++ не определена конкретная реализация ни одного контейнера. Однако поведение и сложность, определенные стандартом, не оставляют большого выбора вариантов. По этой причине на практике реализации различаются лишь незначительными деталями.

[2] Библиотека National Institutes of Health's Class Library (NIHCL) — одна из первых библиотек классов на языке C++.

При выборе подходящего контейнера следует учитывать его возможности, а не порядок следования элементов. Фактически автоматическая сортировка элементов в ассоциативном массиве *не означает*, что эти контейнеры предназначены специально для сортировки элементов. Элементы можно сортировать и в последовательном контейнере. Основное преимущество автоматической сортировки — более высокое быстродействие при поиске элементов. В частности, всегда можно использовать бинарный поиск, имеющий логарифмическую, а не линейную сложность. Это означает, например, что при поиске элемента в коллекции, состоящей из 1000 элементов, в среднем приходится выполнять только 10, а не 500 сравнений (см. раздел 2.2). Таким образом, автоматическая сортировка — это лишь (полезный) "побочный эффект" реализации ассоциативного контейнера, разработанного с целью повышения быстродействия.

В следующих подразделах подробно рассматриваются контейнерные классы: их типичные реализации, а также преимущества и недостатки. В главе 7 подробно описаны точное поведение контейнерных классов, их общие и индивидуальные возможности, а также функции-члены. Использование каждого контейнера продемонстрировано в разделе 7.12.

6.2.1. Последовательные контейнеры

В библиотеке STL определены следующие стандартные последовательные контейнеры.

- Массивы (класс `array`).
- Векторы.
- Деки.
- Списки (одно- и двухсвязные).

Начнем с векторов, поскольку в стандарте языка C++ массивы появились позже, в документе TR1. Кроме того, они имеют особенности, отличающие их от контейнеров библиотеки STL в принципе.

Векторы

Вектор управляет элементами, хранящимися в динамическом массиве. Он обеспечивает произвольный доступ к элементам, т.е. к каждому элементу можно обратиться напрямую по соответствующему индексу. Добавление и удаление элементов происходит в конце массива и выполняется очень быстро[3]. Однако вставка элемента в середину или в начало массива занимает достаточно много времени, потому что все последующие элементы должны переместиться, чтобы освободить место для нового элемента, сохраняя порядок следования.

В следующем примере определяется вектор целых чисел, выполняются вставка шести элементов и вывод элементов вектора на экран:

```
// stl/vector1.cpp
```

[3] Строго говоря, добавление элементов выполняется очень быстро *с учетом амортизации*. Добавление отдельного элемента может оказаться медленным, если вектор должен переместиться в новый участок памяти и скопировать туда все существующие элементы. Однако, поскольку перемещение вектора в новый участок памяти происходит относительно редко, в целом добавление происходит очень быстро. (Вопросы сложности обсуждались в разделе 2.2.)

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

int main()
{
    vector<int> coll;    // вектор для целочисленных элементов

    // добавляем элементы со значениями от 1 до 6
    for (int i=1; i<=6; ++i) {
        coll.push_back(i);
    }

   // выводим на экран все элементы и пробел
   for (int i=0; i<coll.size(); ++i) {
       cout << coll[i] << ' ';
   }
   cout << endl;
}
```

Заголовочный файл для векторов включается с помощью директивы

```
#include <vector>
```

Следующее объявление создает вектор элементов типа `int`:

```
vector<int> coll;
```

Этот вектор не инициализирован никакими значениями, поэтому конструктор, заданный по умолчанию, создает его в виде пустой коллекции. Функция `push_back()` добавляет элемент в контейнер.

```
coll.push_back(i);
```

Эта функция-член есть во всех последовательных контейнерах, в которых возможно добавление элемента за достаточно быстрое время.

Функция-член `size()` возвращает количество элементов, содержащихся в контейнере.

```
for (int i=0; i<coll.size(); ++i) {
   ...
}
```

Функция `size()` есть в любом контейнерном классе, за исключением односвязных списков (класса `forward_list`). Используя операцию индексирования `[]`, можно получить доступ к отдельному элементу вектора.

```
cout << coll[i] << ' ';
```

Здесь элементы выводятся в стандартный выходной поток данных, поэтому результат работы программы выглядит следующим образом:

```
1 2 3 4 5 6
```

Деки

Термин *дек* (deque) является аббревиатурой от "double-ended queue" ("двусторонняя очередь"). Дек — это динамический массив, реализованный таким образом, что он может расти в обоих направлениях. Таким образом, вставка элемента в конец *и* в начало дека выполняется быстро. Однако вставка элемента в середину дека может потребовать больше времени, потому что при этом необходимо перемещать элементы.

В следующем примере объявляется дек, содержащий числа с плавающей точкой, выполняется вставка элементов от 1,1 до 6,6 в начало контейнера и вывод всех элементов дека на экран:

```
// stl/deque1.cpp

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

int main()
{
   deque<float> coll; // дек для чисел с плавающей точкой

   // вставляем элементы от 1.1 до 6.6 в начало дека
   for (int i=1; i<=6; ++i) {
        coll.push_front(i*1.1); // insert at the front
   }

   // выводим на экран все элементы и пробел
   for (int i=0; i<coll.size(); ++i) {
        cout << coll[i] << ' ';
   }
   cout << endl;
}
```

В этом примере заголовочный файл для дека включается директивой

```
#include <deque>
```

Следующее объявление создает пустую коллекцию чисел с плавающей точкой:

```
deque<float> coll;
```

Затем функция `push_front()` вставляет элементы:

```
coll.push_front(i*1.1);
```

Функция `push_front()` вставляет элементы в начало коллекции. Этот вид вставки порождает обратный порядок следования элементов, потому что каждый элемент, вставленный в начал дека, предшествует ранее вставленным элементам. Таким образом, результат работы программы выглядит следующим образом:

```
6.6 5.5 4.4 3.3 2.2 1.1
```

Вставку элементов в дек можно также выполнять с помощью функции-члена `push_back()`. Однако функция `push_front()` не предусмотрена для векторов из-за ее низкой производительности при работе с этим видом контейнеров (при вставке элемента в начало вектора необходимо переместить все его элементы). Обычно контейнеры из библиотеки STL содержат только те функции-члены, которые обеспечивают хорошую производительность, причем под словом "хорошая" в данном случае подразумевается константная или логарифмическая сложность. Это не позволяет программисту вызывать функцию, которая может снизить быстродействие программы.

Тем не менее *можно* вставить элемент в начало вектора — как и в середину вектора и дека, — используя универсальную функцию вставки, которую мы рассмотрим позже.

Массивы

Объект класса `array`[4] управляет своими элементами, хранящимися в массиве фиксированного размера (иногда такие массивы называют статическими или массивами в стиле языка C). Таким образом, изменять можно только значения элементов, но не их количество. Следовательно, размер массива необходимо указывать в момент его создания. Массив также обеспечивает произвольный доступ, т.е. к каждому элементу массива можно обратиться по индексу.

Рассмотрим пример массива, содержащего строки:

```
// stl/array1.cpp

#include <array>
#include <string>
#include <iostream>
using namespace std;

int main()
{
   // массив из 5 строк:
   array<string,5> coll = { "hello", "world" };

   // выводим на экран каждый элемент и его индекс
   for (int i=0; i<coll.size(); ++i) {
        cout << i << ": " << coll[i] << endl;
   }
}
```

Заголовочный файл для массивов включается директивой

```
#include <array>
```

Следующее объявление создает массив из пяти элементов типа `string`:

```
array<string,5> coll
```

По умолчанию эти элементы инициализируются конструктором, предусмотренным по умолчанию для данного типа элементов. Это значит, что элементы фундаментальных типов имеют неопределенное начальное значение.

[4] Класс `array<>` был введен в документе TR1.

Однако в данной программе используется список инициализации (см. раздел 3.1.3), позволяющий инициализировать объекты класса в момент их создания значениями, указанными в списке. В соответствии со стандартом C++11 такой вид инициализации предусмотрен для каждого контейнера, в том числе для векторов и деков. В нашем случае для фундаментального типа используется *инициализация нулем*, т.е. начальное значение данных фундаментального типа всегда равно нулю (см. раздел 3.2.1).

В данной программе мы выводим на экран все элементы и их индексы с помощью функции `size()` и операции индексирования `[ ]`. Результаты работы программы имеют следующий вид:

```
0: hello
1: world
2:
3:
4:
```

Как видим, программа выводит пять строк, поскольку размер массива равен пяти. В соответствии со списком инициализации первые два элемента равны `"hello"` и `"world"`, а остальные элементы имеют значения, заданные по умолчанию, т.е. остаются пустыми строками.

Отметим, что количество элементов является частью типа массива. Таким образом, `array<int,5>` и `array<int,10>` — это два разных типа, поэтому их нельзя присваивать и сравнивать.

Списки

Исторически в стандарте C++98 был только один класс списков. Однако в соответствии со стандартом C++11 библиотека STL содержит два разных типа списков: `list<>` и `forward_list<>`. Таким образом, термин *список* может относиться к конкретному классу или обозначать оба класса списков. Но в некотором смысле однонаправленный список (forward list) — это просто ограниченный список (list), поэтому на практике это различие не так важно. Соответственно, когда мы используем термин *список*, то обычно подразумеваем класс `list<>`, хотя часто этот термин можно применять и к классу `forward_list<>`. Для того чтобы уточнить, какой класс мы имеем в виду, будем называть класс `forward_list<>` однонаправленным списком. Итак, в этом разделе мы рассматриваем обычные списки, которые изначально являлись частью библиотеки STL.

Класс `list<>` реализован как двусвязный список элементов. Это значит, что каждый элемент в списке занимает свой собственный сегмент памяти и ссылается на предыдущий и следующий элементы.

Списки не поддерживают произвольный доступ к элементам. Например, для доступа к десятому элементу списка необходимо пройти девять предыдущих элементов, следуя по цепочке их ссылок. Однако переход к следующему или предыдущему элементу выполняется за константное время. Таким образом, доступ к произвольному элементу имеет линейную временную сложность, потому что среднее расстояние перехода пропорционально количеству элементов. Это намного хуже амортизированного константного времени доступа к элементам векторов и деков.

Преимущество списка заключается в том, что в любой позиции вставка и удаление элемента осуществляются быстро. Для этого достаточно лишь изменить ссылки. Благодаря

этому перемещение элемента в середине списка выполняется очень быстро по сравнению с перемещением элемента в середине вектора или дека.

В следующем примере мы создаем пустой список символов, вставляем в него символы от 'a' до 'z' и выводим все элементы на экран:

```
// stl/list1.cpp

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

int main()
{
   list<char> coll; // список символьных элементов

   // добавляем элементы от 'a' до 'z'
   for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
   }

   // выводим на экран все элементы,
   // используя диапазонный цикл
   for (auto elem : coll) {
        cout << elem << ' ';
   }
   cout << endl;
}
```

Как обычно, для определения коллекции типа `list`, содержащей символы, используется заголовочный файл для списков , `<list>`.

```
list<char> coll;
```

Для вывода всех элементов используется диапазонный цикл `for`, появившийся в стандарте C++11, и позволяющий выполнять операторы для каждого элемента (см. раздел 3.1.4). Прямой доступ к элементам с помощью операции `[ ]` для списков не предусмотрен. Это объясняется тем, что списки не обеспечивают произвольный доступ и операция `[]` имел бы низкое быстродействие.

В цикле используется ключевое слово `auto`, с помощью которого объявляется тип текущего элемента `coll`. Таким образом, тип объекта `elem` автоматически выводится как `char`, потому что `coll` — это коллекция элементов типа `char` (детали вывода типа с помощью ключевого слова `auto` см. в разделе 3.1.2). Вместо этого можно было бы явно объявить тип объекта `elem`.

```
for (char elem : coll) {
   ...
}
```

Обратите внимание на то, что объект `elem` всегда является копией текущего элемента. Таким образом, его можно модифицировать, и это влияло бы только на операторы, выполняемые с этим элементом. В объекте `coll` никакие элементы при этом не модифицируются.

Для того чтобы изменить элементы в переданной коллекции, объект `elem` необходимо объявить как неконстантную ссылку:

```
for (auto& elem : coll) {
   ... // любое изменение объекта elem приводит
       // к изменению текущего элемента в объекте coll
}
```

Для того чтобы избежать копирования при передаче параметра функции, следует использовать константную ссылку. Таким образом, следующая шаблонная функция выводит на экран все элементы переданного ей контейнера:

```
template <typename T>
void printElements (const T& coll)
{
   for (const auto& elem : coll) {
        std::cout << elem << std::endl;
   }
}
```

До появления стандарта C++11 для доступа ко всем элементам следовало использовать итераторы, которые будут рассмотрены позже. Соответствующий пример приведен в разделе 6.3.

Альтернативным способом вывода всех элементов на экран до появления стандарта C++11 (без использования итераторов) был вывод и удаление первого элемента в списке.

```
// stl/list2.cpp

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

int main()
{
   list<char> coll; // список символов

   // добавление элементов от 'a' до 'z'
   for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
   }

   // вывод всех элементов
   // - до полного исчерпания списка
   // - вывод и удаление первого элемента
   while (! coll.empty()) {
          cout << coll.front() << ' ';
          coll.pop_front();
   }
   cout << endl;
}
```

Функция-член `empty()` возвращает результат проверки наличия элементов в контейнере. Цикл продолжается, пока эта функция возвращает значение `false` (т.е. контейнер содержит элементы).

```
while (! coll.empty()) {
   ...
}
```

В цикле функция-член `front()` возвращает первый элемент.

```
cout << coll.front() << ' ';
```

Функция `pop_front()` удаляет первый элемент.

```
coll.pop_front();
```

Отметим, что функция `pop_front()` не возвращает удаленный элемент, поэтому два предыдущих оператора объединить нельзя.

Результат работы программы зависит от используемой кодировки символов. При использовании кодировки ASCII результат имеет следующий вид[5]:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Однонаправленные списки

С появлением стандарта C++11 в стандартную библиотеку языка C++ включен дополнительный контейнер: однонаправленный список. Класс `forward_list<>` реализован как *односвязный* список элементов. Как и в обычном списке `list`, каждый элемент занимает отдельный сегмент памяти, но для экономии памяти он ссылается только на следующий элемент.

В результате однонаправленный список представляет собой ограниченный список, в котором не поддерживаются все операции, предусматривающие перемещение назад или вызывающие снижение быстродействия. По этой причине в этом классе не реализованы такие функции-члены, как `push_back()` и даже `size()`.

На практике это ограничение является еще более неудобным, чем кажется. Одна из проблем заключается в том, что элемент невозможно найти и просто удалить или вставить перед ним другой элемент. Дело в том, что, для того чтобы удалить элемент, необходимо находиться на позиции предыдущего элемента, поскольку именно он ссылается на следующий элемент. В результате в однонаправленных списках предусмотрены специальные функции-члены, которые будут рассмотрены в разделе 7.6.2.

Рассмотрим небольшой пример использования однонаправленного списка:

```
// stl/forwardlist1.cpp

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

int main()
```

[5] Для других кодировок результат работы может содержать символы, которые не являются буквами, или даже оказаться пустым (если `'z'` не больше `'a'`).

```
{
   // создаем однонаправленный список для нескольких простых чисел
   forward_list<long> coll = { 2, 3, 5, 7, 11, 13, 17 };

   // дважды изменяем размер
   // - примечание: низкая производительность
   coll.resize(9);
   coll.resize(10,99);

   // выводим на экран все элементы:
   for (auto elem : coll) {
        cout << elem << ' ';
   }
   cout << endl;
}
```

Как обычно, для работы с однонаправленными списками используется заголовочный файл <forward_list>. Это позволяет нам определить коллекцию типа forward_list для длинных целых чисел, представляющих собой несколько простых чисел.

```
forward_list<long> coll = { 2, 3, 5, 7, 11, 13, 17 };
```

Функция-член resize() изменяет количество элементов. Если размер увеличивается, можно передать дополнительный параметр для указания значений новых элементов. В противном случае используется значение, предусмотренное по умолчанию (нуль для элементарных типов). Отметим, что вызов функции resize() связан с большими затратами. Он имеет линейную сложность, потому что, для того чтобы достичь конца списка, необходимо пройти все элементы один за другим. Однако эту операцию предусматривают практически все последовательные контейнеры, несмотря на ее возможную низкую производительность (этой функции нет только в классе array, потому что в этом классе размер является фиксированным).

Как обычно при работе со списками, для вывода на экран всех элементов используется диапазонный цикл for. Результаты работы программы имеют следующий вид:

```
2 3 5 7 11 13 17 0 0 99
```

6.2.2. Ассоциативные контейнеры

Ассоциативные контейнеры автоматически сортируют свои элементы в соответствии с определенным критерием. Элементы могут быть либо значениями любого типа, либо парой "ключ–значение". Для пар "ключ–значение" каждый ключ, который может иметь любой тип, отображается в ассоциированное с ним значение, которое также может иметь любой тип. Критерий сортировки может задаваться в виде функции, сравнивающей значения или ключи. По умолчанию контейнеры сравнивают элементы или ключи с помощью операции <. Однако программист может создать собственную функцию для сравнения, определив другой критерий сортировки.

Обычно ассоциативные контейнеры реализуются в виде бинарных деревьев. Следовательно, каждый элемент (узел) имеет одного родителя и двух потомков. Все предки слева от узла имеют меньшие значения, а справа — большие. Ассоциативные контейнеры отличаются видами элементов и способами работы с дубликатами.

Основное преимущество ассоциативных контейнеров заключается в том, что поиск элемента с заданным значением выполняется довольно быстро, потому что имеет логарифмическую сложность (во всех последовательных контейнерах поиск имеет линейную сложность). Таким образом, при использовании ассоциативного контейнера, содержащего 1000 элементов, в среднем приходится выполнять 10, а не 500 сравнений. Однако ассоциативные контейнеры имеют недостаток — значения невозможно изменять непосредственно, потому что при этом может быть нарушен автоматический порядок сортировки элементов.

В библиотеке STL определены следующие ассоциативные контейнеры.

- **Множество** — коллекция, в которой элементы сортируются в соответствии со своими значениями. Каждый элемент входит в коллекцию только один раз, так что дубликаты не допускаются.
- **Мультимножество** — это множество, в котором разрешены дубликаты. Таким образом, мультимножество может содержать несколько элементов, имеющих одинаковые значения.
- **Отображение** содержит пары "ключ–значение". У каждого элемента есть ключ, использующийся для сортировки, и значение. Каждый ключ может входить в отображение только один раз, так что дубликаты ключей не разрешаются. Отображение может использоваться в качестве *ассоциативного массива*, т.е. массива, имеющего индекс произвольного типа (см. раздел 6.2.4).
- **Мультиотображение** — это отображение, в котором разрешены дубликаты. Таким образом, мультиотображение может содержать несколько элементов, имеющих одинаковые ключи. Мультиотображение может также использоваться как *словарь* (раздел 7.8.5).

Все перечисленные ассоциативные контейнеры имеют необязательный шаблонный аргумент для критерия сортировки. По умолчанию критерий сортировки использует операцию <. Критерий сортировки используется также для проверки эквивалентности элементов[6]; иначе говоря, два элемента являются дубликатами, если ни одно из их значений или ключей не меньше другого.

Множество можно рассматривать как особую разновидность отображения, в котором значение идентично ключу. Фактически все эти ассоциативные контейнерные типы обычно реализуются с помощью бинарного дерева.

Примеры использования множеств и мультимножеств

Рассмотрим сначала пример использования мультимножества:

```
// stl/multiset1.cpp

#include <set>
#include <string>
#include <iostream>
using namespace std;

int main()
```

[6] Обратите внимание на то, что здесь использован термин "эквивалентные элементы", а не "равные элементы", которые подразумевают применение оператора == к элементу в целом.

```
{
   multiset<string> cities {
      "Braunschweig", "Hanover", "Frankfurt", "New York",
      "Chicago", "Toronto", "Paris", "Frankfurt"
   };

   // вывод на экран каждого элемента:
   for (const auto& elem : cities) {
        cout << elem << " ";
   }
   cout << endl;

   // вставка дополнительных значений:
   cities.insert( {"London", "Munich", "Hanover", "Braunschweig"} );

   // вывод на печать каждого элемента:
   for (const auto& elem : cities) {
        cout << elem << " ";
   }
   cout << endl;
}
```

После объявления типа множества в заголовочном файле `<set>` мы можем объявить контейнер `cities` как мультимножество строк:

```
multiset<string> cities
```

В этом объявлении для инициализации передается набор элементов, которые впоследствии вставляются с помощью списка инициализации (см. раздел 3.1.3). Для печати всех элементов используется диапазонный цикл `for` (см. раздел 3.1.4). Отметим, что элементы объявляются с модификаторами `const auto&`, т.е. мы выводим тип элементов из контейнера (см. раздел 3.1.2) и не создаем копию каждого элемента для обработки в цикле.

Все элементы сортируются автоматически, так что первый вывод программы выглядит следующим образом:

```
Braunschweig Chicago Frankfurt Frankfurt Hanover New York Paris Toronto
```

Второй вывод имеет следующий вид:

```
Braunschweig Braunschweig Chicago Frankfurt Frankfurt Hanover Hanover London
Munich New York Paris Toronto
```

Поскольку мы используем мультимножество, а не множество, дубликаты разрешаются. Если бы мы решили использовать множество, а не мультимножество, то каждое значение выводилось бы на печать только один раз. Если бы мы использовали неупорядоченное мультимножество, то порядок следования элементов оставался бы неопределенным (см. раздел 6.2.3).

Примеры использования отображений и мультиотображений

Следующий пример демонстрирует использование отображений и мультиотображений:

```
// stl/multimap1.cpp

#include <map>
```

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

int main()
{
   multimap<int,string> coll; // контейнер для пар int/string

   // присваиваем некоторые элементы в произвольном порядке
   // - значение с ключом 1 вставляется дважды
   coll = { {5,"tagged"},
            {2,"a"},
            {1,"this"},
            {4,"of"},
            {6,"strings"},
            {1,"is"},
            {3,"multimap"} };

   // выводим на печать значения всех элементов
   // - элемент second является значением
   for (auto elem : coll) {
        cout << elem.second << ' ';
   }
   cout << endl;
}
```

После включения заголовочного файла <map> объявляется отображение с элементами, имеющими ключ типа int и значение типа string.

```
multimap<int,string> coll;
```

Поскольку элементы отображений и мультиотображений представляют собой пары "ключ–значение", объявление, вставка и доступ к элемента немного отличаются.

- Во-первых, для инициализации (присвоения или вставки) элементов необходимо передавать пары "ключ–значение" с помощью списков инициализации. Внутренние списки определяют ключи и значение каждого элемента; внешние списки группируют все эти элементы. Таким образом, инструкция {5,"tagged"} описывает вставку первого элемента.
- При обработке элементов мы снова работаем с парами "ключ–значение". Фактически типом элемента является класс pair<const *ключ, значение*> (тип pair введен в разделе 5.1.1). Ключ является константой, потому что любая модификация его значения может нарушить порядок следования элементов, автоматически упорядоченных контейнером. Поскольку объекты структуры pair не имеют операции вывода, их невозможно вывести на экран целиком. Вместо этого необходимо получить доступ к членам структуры pair, которые называются first и second.

Таким образом, следующее выражение создает вторую часть пары "ключ–значение", т.е. значение элемента мультиотображения:

```
elem.second
```

Аналогично следующее выражение создает первую часть пары "ключ–значение", т.е. ключ элемента мультиотображения:

```
elem.first
```

В результате программа выводит на экран такую строку:

```
this is a multimap of tagged strings
```

До появления стандарта C++11 не было строгой гарантии сохранения порядка эквивалентных элементов (т.е. элементов, имеющих одинаковые ключи). Итак, до принятия стандарта C++11порядок следования строк `"this"` и `"is"` мог быть любым. Стандарт C++11 гарантирует, что новые элементы вставляются после эквивалентных элементов, которые уже содержатся в мультимножествах и мультиотображениях. Кроме того, при вызове функций `insert()`, `emplace()` или `erase()` порядок эквивалентных элементов сохраняется.

Другие примеры ассоциативных контейнеров

В разделе 6.2.4 приведен пример использования отображения, которое называется *ассоциативным массивом*.

В разделе 7.7 подробно обсуждаются множества и мультимножества и приводятся дополнительные примеры. В разделе 7.8 подробно рассматриваются отображения и мультиотображения с примерами.

Мультиотображения можно использовать как *словари* (см. раздел 7.8.5).

6.2.3. Неупорядоченные контейнеры

В неупорядоченном контейнере элементы не имеют определенного порядка. Следовательно, если вставить три элемента, то при обходе контейнера они могут следовать в любом порядке. Если вставить четвертый элемент, то порядок следования ранее вставленных элементов может измениться. Единственным фактом, имеющим значение, является то, что конкретный элемент находится *где-то* в контейнере. Даже если два контейнера содержат одинаковые элементы, их порядок может быть разным. Представьте себе, что это просто мешок.

Неупорядоченные контейнеры обычно реализуются в виде хеш-таблицы (рис. 6.3). Таким образом, по существу, контейнер — это массив связанных списков. Позиция элемента в массиве вычисляется с помощью *хеш-функции*. Цель заключается в том, чтобы каждый элемент имел свою позицию, обеспечивающую к нему быстрый доступ, при условии быстрого вычисления хеш-функции. Однако, поскольку быстрые идеальные хеш-функции не всегда возможны или могут потребовать огромный объем памяти для массива, разрешается хранить в одной и той же позиции несколько элементов. По этой причине элементами массива являются связанные списки, позволяющие хранить в одной позиции массива несколько элементов контейнера.

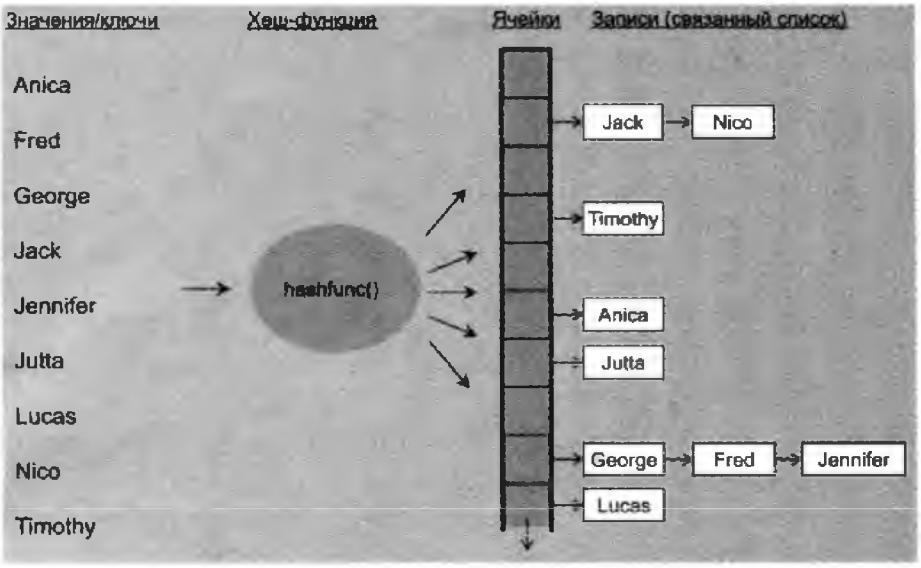


Рис. 6.3. Неупорядоченные массивы в виде хеш-таблицы

Основное преимущество неупорядоченных контейнеров заключается в том, что поиск элемента, имеющего конкретное значение, выполняется еще быстрее, чем в ассоциативном массиве. Фактически использование неупорядоченных контейнеров обеспечивает амортизированную константную сложность, при условии, что хеш-функция является достаточно хорошей. Однако создать хорошую хеш-функцию нелегко (см. раздел 7.9.2), так что может понадобиться много памяти для ячеек.

Аналогично ассоциативным контейнерам, в библиотеке STL предусмотрены следующие неупорядоченные контейнеры.

- **Неупорядоченное множество** — коллекция неупорядоченных элементов, каждый из которых может храниться только в одном экземпляре. Таким образом, дубликаты запрещены.
- **Неупорядоченное мультимножество** — это неупорядоченное множество, в котором разрешены дубликаты. Таким образом, неупорядоченное мультимножество может содержать несколько элементов, имеющих одинаковые значения.
- **Неупорядоченное отображение** — контейнер, содержащий пары "ключ–значение". Каждый ключ может храниться только в одном экземпляре, так что дубликаты ключей запрещены. Неупорядоченное отображение можно использовать в качестве *ассоциативного массива*, т.е. массива с индексом произвольного типа (см. раздел 6.2.4).
- **Неупорядоченное мультиотображение** — это неупорядоченное отображение, в котором разрешены дубликаты. Таким образом, неупорядоченное мультимножество может содержать несколько элементов с одинаковыми ключами. Неупорядоченное мультимножество моно использовать в качестве *словаря* (см. раздел 7.9.7).

Все эти классы неупорядоченных контейнеров имеют несколько необязательных шаблонных аргументов, задающих хеш-функцию и критерий эквивалентности. Критерий эквивалентности используется для поиска конкретных значений и идентификации дубликатов. По умолчанию критерием эквивалентности является операцию ==.

Неупорядоченное множество можно рассматривать как разновидность неупорядоченного отображения, в котором значение идентично ключу. Обычно все эти типы неупорядоченных контейнеров создаются на основе одной и той же базовой реализации хеш-таблицы.

Примеры использования неупорядоченных множеств и мультимножеств

В первом примере демонстрируется использование неупорядоченного мультимножества строк.

```
// stl/unordmultiset1.cpp

#include <unordered_set>
#include <string>
#include <iostream>
using namespace std;

int main()
{
   unordered_multiset<string> cities {
      "Braunschweig", "Hanover", "Frankfurt", "New York",
      "Chicago", "Toronto", "Paris", "Frankfurt"
   };

   // выводим на печать каждый элемент:
   for (const auto& elem : cities) {
        cout << elem << " ";
   }
   cout << endl;

   // вставляем дополнительные значения:
   cities.insert( {"London", "Munich", "Hanover", "Braunschweig"} );

   // выводим на печать каждый элемент:
   for (const auto& elem : cities) {
        cout << elem << " ";
   }
   cout << endl;
}
```

После включения требуемого заголовочного файла

```
#include <unordered_set>
```

можно объявить и инициализировать неупорядоченное множество строк:

```
unordered_multiset<string> cities { ... };
```

Если теперь вывести на печать все элементы, то порядок может оказаться другим, потому что он не определен. Гарантируется лишь, что дубликаты, которые разрешаются в *мультимножестве*, будут сгруппированы в порядке их вставки. Таким образом, один из возможных вариантов вывода может выглядеть следующим образом:

```
Paris Toronto Chicago New York Frankfurt Frankfurt Hanover Braunschweig
```

Любая вставка может изменить этот порядок. На самом деле этот порядок может изменить любая операция, связанная с повторным хешированием. Таким образом, после вставки нескольких дополнительных значений результат может быть таким:

```
London Hanover Hanover Frankfurt Frankfurt New York Chicago Munich Braunschweig
Braunschweig Toronto Paris
```

Последствия зависят от стратегии повторного хеширования, на которую частично может влиять программист. Например, можно зарезервировать достаточно много памяти, чтобы для вставки определенного количества элементов повторного хеширования не потребовалось. Кроме того, для того чтобы иметь возможность удалять элементы при их обработке, стандарт гарантирует, что удаление не требует повторного хеширования. Однако вставка после удаления может привести к повторному хешированию. Подробнее об этом в разделе 7.9.

В принципе ассоциативные и неупорядоченные контейнеры имеют одинаковые интерфейсы, отличаться могут лишь объявления. При этом неупорядоченные контейнеры предоставляют специальные функции-члены, позволяющие влиять на внутреннее поведение или инспектировать текущее состояние. Таким образом, в представленном примере лишь заголовочные файлы и типы отличаются от соответствующего примера, демонстрирующего использование обычного мультимножества в разделе 6.2.2.

До появления стандарта C++11 для доступа к элементам требовались итераторы. Пример приведен в разделе 6.3.1.

Примеры использования неупорядоченных отображений и мультиотображений

Пример, демонстрирующий использование мультиотображений, можно распространить и на неупорядоченное мультиотображение, если заменить название `map` на `unordered_map` в директиве `include` и название `multimap` на `unordered_multimap` в объявлении контейнера.

```
#include <unordered_map>
...
unordered_multimap<int,string> coll;
...
```

Единственное отличие заключается в том, что порядок следования элементов не определен. Однако на большинстве платформ элементы остаются упорядоченными, поскольку в качестве хеш-функции по умолчанию используется операция деления по модулю. Таким образом, неопределенность порядка не означает его отсутствие. В то же время это свойство не гарантируется, и если добавить несколько новых элементов, то порядок может измениться.

Рассмотрим другой пример использования неупорядоченного отображения. В данном случае мы используем неупорядоченное отображение, в котором ключами являются строки, а значениями — числа типа `double`.

```
// stl/unordmap1.cpp

#include <unordered_map>
#include <string>
```

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

int main()
{
   unordered_map<string,double> coll { { "tim", 9.9 },
                                       { "struppi", 11.77 }
                                     };

   // возводим значение каждого элемента в квадрат:
   for (pair<const string,double>& elem : coll) {
        elem.second *= elem.second;
   }

   // выводим на печать каждый элемент (ключ и значение):
   for (const auto& elem : coll) {
        cout << elem.first << ": " << elem.second << endl;
   }
}
```

После обычных включений заголовочных файлов для отображений, строк и потоков ввода-вывода объявляется неупорядоченное отображение, которое инициализируется двумя элементами. Здесь используются вложенные списки инициализации, так что отображение инициализируется двумя элементами:

```
{ "tim", 9.9 }
```

и

```
{ "struppi", 11.77 }
```

Затем мы возводим значение каждого элемента в квадрат.

```
for (pair<const string,double>& elem : coll) {
     elem.second *= elem.second;
}
```

Здесь можно увидеть внутренний тип элементов — объектов типа `pair<>` (см. раздел 5.1.1), состоящих из константной строки и чисел типа `double`. Таким образом, модифицировать ключ `first` в элементе невозможно:

```
for (pair<const string,double>& elem : coll) {
     elem.first = ...; // ОШИБКА: ключи отображения являются константами
}
```

В соответствии со стандартом C++11 явно указывать тип элементов необязательно, поскольку в диапазонном цикле `for` он выводится из типа контейнера. По этой причине во втором цикле, который выводит все элементы на печать, используется ключевое слово `auto`. Фактически, объявляя `elem` как `const auto&`, мы избегаем создания копий:

```
for (const auto& elem : coll) {
     cout << elem.first << ": " << elem.second << endl;
}
```

В результате один из *возможных* результатов работы программы может выглядеть следующим образом:

```
struppi: 138.533
tim: 98.01
```

Этот порядок не гарантирован, поскольку реальный порядок не определен. Если бы использовался обычный контейнер `map`, то порядок элементов был бы гарантированным и элемент с ключом `"struppi"` выводился бы на экран до элемента с ключом `"tim"`, потому что отображение сортирует элементы по ключу, а строка `"struppi"` предшествует строке `"tim"`. Пример использования отображения и применения алгоритмов и лямбда-функций вместо диапазонного цикла `for` приведен в разделе 7.8.5.

Другие примеры неупорядоченных контейнеров

Классы для неупорядоченных контейнеров имеют несколько дополнительных необязательных шаблонных аргументов, задающих хеш-функцию и критерий эквивалентности. Для фундаментальных типов и строк предусмотрена хеш-функция по умолчанию, а для других типов необходимо определять свою собственную хеш-функцию. Эта тема обсуждается в разделе 7.9.2.

В следующем разделе приводится пример использования отображения в качестве ассоциативного массива. Неупорядоченные контейнеры вместе с дополнительными примерами рассматриваются в разделе 7.9. Неупорядоченные мультиотображения также могут использоваться как *словари* (см. раздел 7.9.7).

6.2.4. Ассоциативные массивы

Отображения и неупорядоченные отображения являются коллекциями пар "ключ–значение" с уникальными ключами. Такие коллекции можно интерпретировать как *ассоциативный массив*, т.е. массив, индекс которого не является целым числом. В результате оба контейнера допускают использование операции индексирования `[ ]`.

Рассмотрим следующий пример:

```
// stl/assoarray1.cpp

#include <unordered_map>
#include <string>
#include <iostream>
using namespace std;

int main()
{
   // тип контейнера:
   // - unordered_map: элементами являются пары ключ-значение
   // - string: ключи имеют тип string
   // - float: значения имеют тип float
   unordered_map<string,float> coll;

   // вставляем элементы в коллекцию
   // - используя синтаксис ассоциативного массива
   coll["VAT1"] = 0.16;
```

```
    coll["VAT2"] = 0.07;
    coll["Pi"] = 3.1415;
    coll["an arbitrary number"] = 4983.223;
    coll["Null"] = 0;

    // изменяем значение
    coll["VAT1"] += 0.03;

    // выводим на печать разности между значениями VAT
    cout << "VAT difference: " << coll["VAT1"] - coll["VAT2"] << endl;
}
```

Объявление контейнерного типа должно задавать тип ключа и значения:

```
unordered_map<string,float> coll;
```

Это объявление означает, что ключами являются строки, а связанные с ними значения являются числами с плавающей точкой.

В соответствии с концепцией ассоциативных массивов доступ к элементам обеспечивает операция индексирования `[ ]`. Отметим, однако, что эта операция индексирования не похожа на обычное индексирование массивов: если заданному индексу не соответствует ни один элемент, то это *не* ошибка. Новый индекс (или ключ) инициирует создание и вставку нового элемента отображения, у которого данный индекс является ключом. Таким образом, индекс никогда не бывает неправильным.

Следовательно, в нашем примере оператор

```
coll["VAT1"] = 0.16;
```

создает новый элемент, имеющий ключ `"VAT1"` и значение `0.16`.

Следующее выражение создает новый элемент, имеющий ключ `"VAT1"`, и инициализирует его типом, заданным по умолчанию (используя конструктор по умолчанию или нуль для фундаментальных типов данных):

```
coll["VAT1"]
```

Выражение в целом обеспечивает доступ к значению нового элемента, так что оператор присваивания присваивает ему `0.16`.

В соответствии со стандартом C++11 можно в качестве альтернативы использовать функцию `at()` для доступа к значениям элементов по переданному ключу. Если ключ не будет найден, будет сгенерировано исключение `out_of_range`.

```
coll.at("VAT1") = 0.16; // исключение out_of_range при отсутствии элемента
```

Такие выражения, как

```
coll["VAT1"] += 0.03;
```

или

```
coll["VAT1"] - coll["VAT2"]
```

обеспечивают чтение и запись значений указанных элементов. Таким образом, результат программы будет иметь следующий вид:

```
VAT difference: 0.12
```

Как обычно, разница между неупорядоченным и обычным отображением заключается в том, что элементы в неупорядоченном отображении следуют в произвольном порядке, в то время как элементы в отображении упорядочены. Однако доступ к элементу в неупорядоченном отображении имеет амортизированную константную сложность, а в отображении — логарифмическую сложность. По этой причине обычно следует предпочитать неупорядоченные отображения, если только вам не требуется сортировка или вы не можете использовать неупорядоченное отображение из-за того, что окружение не поддерживает свойства стандарта C++11. В этом случае следует просто изменить тип контейнера: удалите префикс "`unordered_`" в директиве `include` и объявлении контейнера.

Использование контейнеров `map` и `unordered_map` в качестве ассоциативных массивов более подробно рассматривается в разделах 7.8.3 и 7.9.5.

6.2.5. Другие контейнеры

Строки

Строки также можно использовать в качестве STL контейнеров. Под *строками* подразумеваются объекты классов строк в языке C++ (`basic_string<>`, `string`, и `wstring`), которые описываются в главе 13. Строки похожи на векторы, но в качестве элементов содержат символы. Детали описаны в разделе 13.2.14.

Обычные массивы в стиле языка C

Другой вид контейнеров относится скорее к ядру языков C и C++, а не к классам. Это обычный массив ("массив в стиле C"), который объявляется с фиксированным или динамическим размером, управляемым функциями `malloc()` и `realloc()`. Однако обычные массивы не относятся к STL-контейнерам, поскольку они не имеют функции-члены, такие как `size()` и `empty()`. Тем не менее архитектура библиотеки STL позволяет применять к ним алгоритмы.

Использование обычных массивов не представляет собой ничего нового. Новым является применение к ним алгоритмов. Эта тема рассматривается в разделе 7.10.2.

В языке C++ больше не обязательно программировать массивы в стиле языка C. Векторы и объекты класса `array` обеспечивают все свойства обычных массивов, но являются более безопасными и удобными. Детали изложены в разделах 7.2.3 и 7.3.3.

Пользовательские контейнеры

В принципе программист может снабдить любой объект со свойствами контейнера интерфейсом, соответствующим требованиям библиотеки STL, что позволит обходить элементы или выполнять стандартные операции над их содержимым. Например, можно написать класс для каталога, в котором можно обходить файлы и манипулировать ими. Наилучшими кандидатами на включение в STL-подобные интерфейсы являются обычные операции над контейнерами, описанные в разделе 7.1.

Однако некоторые объекты, напоминающие контейнеры, не соответствуют концепции библиотеки STL. Например, тот факт, что каждый контейнер в библиотеке STL имеет начало и конец, затрудняет реализацию циклических контейнеров, таких как кольцевой буфер, в стиле библиотеки STL.

6.2.6. Адаптеры контейнеров

Кроме основных контейнерных классов, в стандартной библиотеке C++ предусмотрены *адаптеры контейнеров*, представляющие собой стандартные контейнеры, предоставляющие ограниченный интерфейс для специальных нужд. Эти адаптеры контейнеров реализуются на основе фундаментальных контейнерных классов. Стандартные адаптеры контейнеров перечислены в следующем списке.

- **Стек** (название говорит само за себя) управляет своими элементами по принципу LIFO (последним вошел — первым вышел).
- **Очередь** управляет своими элементами по принципу FIFO (первым вошел — первым вышел). Иначе говоря, это обычный буфер.
- **Очередь с приоритетами** — это контейнер, в котором элементы имеют разные приоритеты. Приоритет зависит от критерия сортировки, который может задавать программист (по умолчанию используется операция <). Очередь с приоритетами по существу представляет собой буфер, в котором очередной элемент всегда имеет наивысший приоритет в очереди. Если наивысший приоритет имеют несколько элементов, то порядок следования этих элементов не определен.

Адаптеры контейнеров исторически являются частью библиотеки STL. Однако с точки зрения программирования они представляют собой всего лишь специальную категорию контейнерных классов, использующих общую архитектуру контейнеров, итераторов и алгоритмов библиотеки STL. По этой причине адаптеры контейнеров описываются отдельно от ядра библиотеки STL в главе 12.

6.3. Итераторы

В соответствии со стандартом C++11 все элементы контейнера можно обойти с помощью диапазонной операции `for`. Однако, для того чтобы найти элемент, обходить все элементы не обязательно. Вместо этого следует перебирать все элементы, пока не будет найден искомый. Кроме того, его позицию, возможно, желательно где-то сохранить, например, чтобы продолжить обход впоследствии. Таким образом, необходима концепция объекта, представляющего позицию элемента в контейнере. Такая концепция существует. Объекты, которые ее реализуют, называются **итераторами**. Как будет показано далее, диапазонный цикл `for` предоставляет удобный интерфейс для реализации этой концепции. Иначе говоря, он изначально использует итераторы для обхода всех элементов контейнера.

Итератор — это объект, который перебирает все элементы (переходит от одного элемента к другому). Он может обойти все элементы контейнера STL или их подмножество. Итератор представляет определенную позицию в контейнере. Для итератора определены следующие фундаментальные операции.

- **Операция** * возвращает элемент, стоящий в текущей позиции. Если этот элемент имеет члены, то с помощью операции -> можно получить доступ к ним непосредственно из итератора.
- **Операция** ++ перемещает итератор вперед на следующий элемент. Большинство итераторов также позволяют возвращение к предыдущему элементу с помощью операции --.
- **Операции** == и != возвращают результат проверки, представляют ли два итератора одну и ту же позицию.
- **Операция** = присваивает итератор (позицию элемента, на которую он ссылается).

Эти операции используют интерфейс, точно совпадающий с интерфейсом обычных указателей в языках C и C++, с помощью которых можно обойти элементы в обычном массиве. Разница заключается в том, что итератор является *интеллектуальным указателем*, т.е. может обходить более сложные структуры данных. Внутреннее поведение итераторов зависит от структуры данных, по которой они перемещаются. По этой причине каждый контейнерный тип предусматривает свой собственный вид итераторов. В результате итераторы имеют общий интерфейс, но разные типы. Это непосредственно приводит к концепции обобщенного программирования: операции используют одинаковый интерфейс, но имеют разные типы, поэтому можно использовать шаблоны для формулировки обобщенных операций, которые применяются к произвольным типам, удовлетворяющим указанному интерфейсу.

Все контейнерные классы имеют одинаковые основные функции-члены, позволяющие перемещать итераторы по элементам контейнера. Наиболее важными из них являются следующие.

- Функция **begin()** возвращает итератор, представляющий начало контейнера, т.е. позицию первого элемента, если таковой имеется в контейнере.
- Функция **end()** возвращает итератор, представляющий конец контейнера, т.е. позицию, *следующую за* последним элементом. Такой итератор называется *запредельным* (past-the-end iterator).

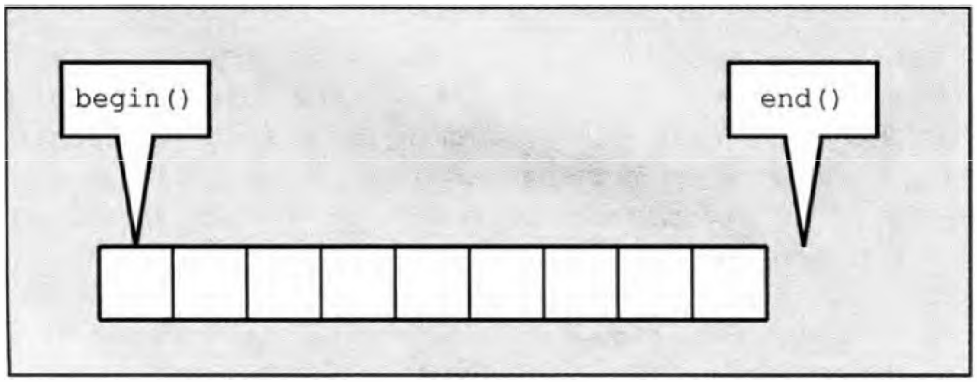


Рис. 6.4. Позиции **begin()** *и* **end()**

Таким образом, функции-члены begin() и end() определяют *полуоткрытый диапазон* (half-open range), включающий первый элемент и не содержащий последний (рис. 6.4). Полуоткрытый *диапазон* имеет два преимущества.

1. Существует простой критерий остановки цикла при обходе всех элементов: цикл продолжается, пока не будет достигнута позиция `end()`.
2. Он позволяет избежать специальной обработки пустых диапазонов. Для пустых диапазонов позиция `begin()` совпадает с позицией `end()`.

Следующий пример демонстрирует использование итераторов для вывода на экран всех элементов списка (это вариант примера из раздела 6.2.1, но с использованием итераторов):

```
// stl/list1old.cpp

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

int main()
{
   list<char> coll; // список символов

   // добавляем элементы от 'a' до 'z'
   for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
   }

   // выводим на печать все элементы:
   // - обходим все элементы
   list<char>::const_iterator pos;
   for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << *pos << ' ';
   }
   cout << endl;
}
```

Снова после создания списка и заполнения его символами от `'a'` до `'z'` мы выводим на экран все элементы. Однако вместо диапазонного цикла `for`:

```
for (auto elem : coll) {
    cout << elem << ' ';
}
```

теперь все элементы выводятся в обычном цикле с помощью итераторов, обходящих элементы контейнера:

```
list<char>::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
     cout << *pos << ' ';
}
```

Итератор `pos` объявляется прямо перед циклом. Он имеет тип итератора для доступа к константным элементам контейнерного класса.

```
list<char>::const_iterator pos;
```

В каждом контейнере объявляются два типа итераторов.

1. Итератор *контейнер*::iterator перемещается по элементам в режиме чтения/записи.
2. Итератор *контейнер*::const_iterator перемещается по элементам только в режиме чтения.

Например, в классе list определения могут иметь следующий вид:

```
namespace std {
   template <typename T>
   class list {
      public:
         typedef ... iterator;
         typedef ... const_iterator;
         ...
   };
}
```

Точный тип iterator и const_iterator определяется реализацией.

В цикле for итератор pos инициализируется позицией первого элемента:

```
pos = coll.begin()
```

Цикл продолжается, пока итератор pos не достигнет конца контейнера:

```
pos != coll.end()
```

Здесь итератор pos сравнивается с так называемым запредельным итератором, представляющим позицию, следующую за последним элементом. Когда цикл выполняет операцию инкремента ++pos, итератор pos перемещается на следующий элемент.

Итератор pos перемещается от первого элемента до последнего (рис. 6.5). Если контейнер не содержит элементов, то тело цикла не выполняется, потому что coll.begin() равно coll.end().

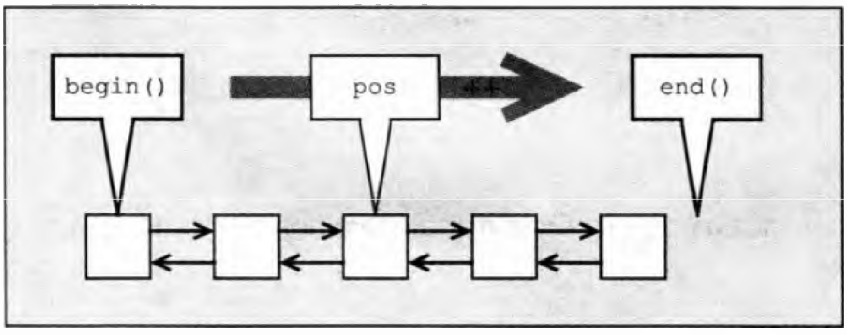


Рис. 6.5. Итератор pos *перемещается по элементам списка*

В теле цикла выражение *pos представляет текущий элемент. В данном примере он записывается в стандартный поток вывода cout, а за ним следует символ пробела. Элементы модифицировать нельзя, поскольку используется тип итератора const_iterator. Таким образом, с точки зрения итератора элемент является константой. Однако, если использовать неконстантный итератор и неконстантный тип элементов, то можно изменять значения элементов. Рассмотрим пример:

```
// переводим все символы в списке в верхний регистр
list<char>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
*pos = toupper(*pos);
}
```

Если итераторы используются для обхода элементов (неупорядоченных) отображений и мультиотображений, то итератор `pos` ссылается на пары "ключ–значение". Таким образом, выражение

```
pos->second
```

возвращает вторую часть пары "ключ–значение", т.е. значение элемента, а выражение

```
pos->first
```

возвращает (константный) ключ.

Сравнение операций ++pos и pos++

Отметим, что для перемещения итератора на следующий элемент здесь используется операция префиксного инкремента ++. Причина заключается в том, что эта операция теоретически обеспечивает более высокое быстродействие по сравнению с постфиксной операцией инкремента. Постфиксная операция неявно использует временный объект, потому что она должна возвращать старую позицию итератора. По этой причине, вообще говоря, `++pos` предпочтительнее `pos++`. Таким образом, следующей версии следует избегать:

```
for (pos = coll.begin(); pos != coll.end(); pos++) {
                                            ^^^^^ // OK,
                                  // но немного медленнее
   ...
}
```

Реально такие способы повышения производительности программы почти никогда не достигают цели, поэтому не следует интерпретировать наши рекомендации слишком буквально для достижения микроскопических преимуществ. Читабельность и удобство сопровождения программ намного важнее, чем оптимизация их быстродействия. Следует также подчеркнуть, что в данном случае выбор префиксной формы инкремента и отказ от постфиксной не связан ни с какими дополнительными расходами. Таким образом, рекомендация использовать префиксные формы инкремента и декремента — просто добрый совет.

Функции cbegin() и cend()

В соответствии со стандартом C++11 ключевое слово `auto` (см. раздел 3.1.2) позволяет указать точный тип итератора (при условии, что итератор был инициализирован во время объявления, так что его тип можно вывести из его начального значения). Таким образом, непосредственная инициализация итератора с помощью функции `begin()` позволяет использовать ключевое слово `auto` для объявления его типа:

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
     cout << *pos << ' ';
}
```

Легко видеть, что использование ключевого слова `auto` делает код более компактным. Без ключевого слова `auto` объявление итератора в цикле выглядело бы следующим образом:

```
for (list<char>::const_iterator pos = coll.begin();
     pos != coll.end(); ++pos) {
     cout << *pos << ' ';
}
```

Другое преимущество применения `auto` заключается в том, что цикл является устойчивым к изменениям кода, таким как модификация типа контейнера. Однако у такой конструкции есть недостаток — итератор теряет свою константность, т.е. появляется риск непреднамеренного присваивания. Выражение

```
auto pos = coll.begin()
```

делает итератор `pos` неконстантным, потому что функция `begin()` возвращает объект типа *контейнер*`::iterator`. Для того чтобы сохранить константность итератора, в стандарте C++11 предусмотрены функции `cbegin()` и `cend()`. Они возвращают объект типа *контейнер*`::const_iterator`.

Резюмируя, отметим, что в стандарте C++11 цикл, который позволяет обходить все элементы контейнера без использования диапазонного цикла `for`, может иметь следующий вид:

```
for (auto pos = coll.cbegin(); pos != coll.cend(); ++pos) {
     ...
}
```

Диапазонный цикл `for` и итераторы

Введя понятие итератора, мы можем объяснить точное поведение диапазонного цикла `for` (range-based for). Для контейнеров диапазонный цикл `for` просто предоставляет удобный интерфейс, позволяющий обойти все элементы переданного диапазона или коллекции, и ничего больше. В каждом цикле реальный элемент инициализируется значением, на которое ссылается текущий итератор.

Таким образом, конструкция

```
for (type elem : coll) {
   ...
}
```

интерпретируется как

```
for (auto pos=coll.begin(), end=coll.end(); pos!=end; ++pos) {
     type elem = *pos;
     ...
}
```

Теперь становится понятным, почему объект `elem` должен объявляться как константная ссылка, чтобы избежать ненужного копирования. В противном случае объект `elem` инициализировался бы копией значения `*pos`. (Подробности см. в разделе 3.1.4.)

6.3.1. Дополнительные примеры использования ассоциативных и неупорядоченных контейнеров

Теперь, получив представление об итераторах, можно привести несколько примеров программ, использующих ассоциативные контейнеры без использования таких языковых конструкций стандарта C++11, как диапазонный цикл `for`, ключевое слово `auto` и списки инициализации. Кроме того, используемые здесь конструкции в некоторых ситуациях могут оказаться полезными и в сочетании со стандартом C++11.

Использование множества до появления стандарта C++11

Первый пример демонстрирует вставку элементов в множество и применение итераторов без использования языковых средств из стандарта C++11.

```
// stl/set1.cpp

#include <set>
#include <iostream>

int main()
{
   // тип коллекции
   typedef std::set<int> IntSet;

   IntSet coll; // объявляем множество для целых чисел

   // вставляем элементы от 1 до 6 в произвольном порядке
   // - обратите внимание на два вызова функции insert() со значением 1
   coll.insert(3);
   coll.insert(1);
   coll.insert(5);
   coll.insert(4);
   coll.insert(1);
   coll.insert(6);
   coll.insert(2);

   // выводим на печать все элементы
   // - обходим все элементы
   IntSet::const_iterator pos;
   for (pos = coll.begin(); pos != coll.end(); ++pos) {
        std::cout << *pos << ' ';
   }
   std::cout << std::endl;
}
```

Как обычно, директива `include` определяет все необходимые типы и операции над множествами.

```
#include <set>
```

Тип контейнера используется в нескольких местах, поэтому сначала определяем его сокращенное имя.

```
typedef set<int> IntSet;
```

Эта инструкция определяет тип `IntSet` как множество элементов типа `int`. Этот тип использует критерий сортировки, используемый по умолчанию, который упорядочивает элементы с помощью операции <, так что элементы следуют в возрастающем порядке. Для того чтобы упорядочить элементы в убывающем порядке или использовать совершенно другой критерий сортировки, программист может передать его в качестве второго шаблонного параметра. Например, следующая инструкция определяет тип множества, в котором элементы упорядочены по убыванию:

```
typedef set<int,greater<int>> IntSet;
```

Объект `greater<>` — это стандартный функциональный объект, который рассматривается в разделе 6.10.2. Критерий сортировки, использующий только часть данных из объекта, например его идентификатор, описан в разделе 10.1.1.

Все ассоциативные контейнеры предусматривают функцию-член `insert()` для вставки нового элемента.

```
coll.insert(3);
coll.insert(1);
...
```

В соответствии со стандартом C++11 можно написать просто:

```
coll.insert ( { 3, 1, 5, 4, 1, 6, 2 } );
```

Каждый вставленный элемент автоматически занимает правильную позицию в соответствии с критерием сортировки. Функции `push_back()` или `push_front()` использовать нельзя, так как они предназначены для последовательных контейнеров. В данном контексте они не имеют смысла, поскольку невозможно указать позицию нового элемента.

На рис. 6.6 показано состояние контейнера после вставки всех значений. Элементы упорядочены во внутренней структуре контейнера, представляющей собой дерево, поэтому в соответствии с текущим критерием сортировки значение левого потомка элемента всегда меньше, а значение правого потомка всегда больше значения родительского элемента. Дубликаты в множестве не допускаются, поэтому контейнер содержит только одну единицу.

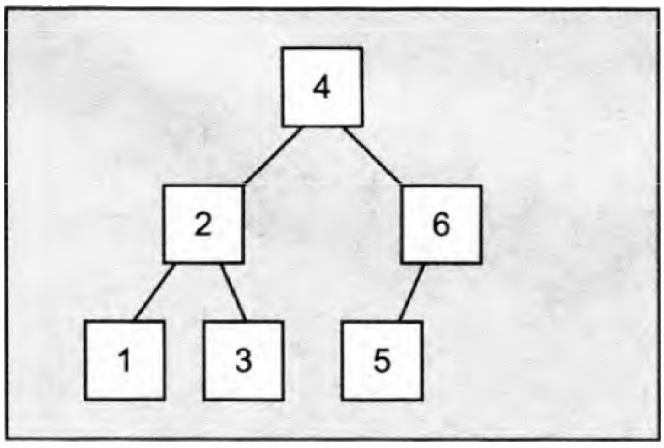


Рис. 6.6. Множество, состоящее из шести элементов

Для того чтобы вывести на экран все элементы контейнера, используем цикл из предыдущего примера, посвященного списку. Итератор обходит все элементы и выводит их на печать.

```
IntSet::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
     cout << *pos << ' ';
}
```

Поскольку итератор определяется контейнером, цикл работает правильно, даже если внутренняя структура контейнера является более сложной. Например, если итератор ссылается на третий элемент, то операция ++ перемещает его к четвертому элементу, находящемуся на вершине. После следующего выполнения операции ++ итератор ссылается на пятый элемент, находящийся внизу (рис. 6.7). Результат работы программы выглядит следующим образом:

```
1 2 3 4 5 6
```

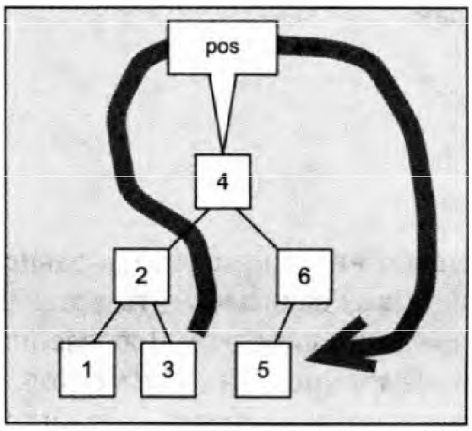


Рис. 6.7. Итератор pos *перемещается по элементам множества*

Для того чтобы использовать мультимножество, а не множество, достаточно изменить тип контейнера; заголовочный файл остается прежним.

```
typedef multiset<int> IntSet;
```

Мультимножество допускает дубликаты, поэтому в нем будут храниться два элемента со значением 1. Таким образом, результат работы программы изменится.

```
1 1 2 3 4 5 6
```

Подробности использования неупорядоченного мультимножества

В другом примере демонстрируется, что произойдет, если выполнить обход всех элементов неупорядоченного мультимножества

```
// stl/unordmultiset2.cpp

#include <unordered_set>
#include <iostream>

int main()
{
   // неупорядоченное мультимножество с целыми значениями
```

```
    std::unordered_multiset<int> coll;

    // вставляем несколько элементов
    coll.insert({1,3,5,7,11,13,17,19,23,27,1});

    // выводим на экран все элементы
    for (auto elem : coll) {
        std::cout << elem << ' ';
    }
    std::cout << std::endl;

    // вставляем еще один элемент
    coll.insert(25);

    // еще раз выводим на экран все элементы
    for (auto elem : coll) {
        std::cout << elem << ' ';
    }
    std::cout << std::endl;
}
```

Порядок следования элементов не определен. Он зависит от внутренней структуры хеш-таблицы и ее хеш-функции. Даже если элементы вставлялись по порядку, в контейнере они располагаются в произвольном порядке[7]. Добавление еще одного элемента может изменить порядок всех имеющихся в множестве элементов.

Таким образом, один из возможных вариантов результата работы программы может выглядеть следующим образом:

```
11 23 1 1 13 3 27 5 17 7 19
23 1 1 25 3 27 5 7 11 13 17 19
```

Как видим, порядок следования элементов действительно не определен, так что если читатель запустит эту программу на своей платформе, то этот порядок может оказаться другим. Добавление одного элемента может изменить порядок следования всех элементов. Однако гарантируется, что элементы с одинаковыми значениями будут расположены рядом.

Обход элементов и вывод их на экран

```
for (auto elem : coll) {
    std::cout << elem << ' ';
}
```

эквивалентен следующему коду:

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    auto elem = *pos;
    std::cout << elem << ' ';
}
```

[7] Отметим, что порядок может оказаться правильным просто потому, что он является одним из возможных вариантов. Если вставить числа 1, 2, 3, 4 и 5 , обычно так и бывает.

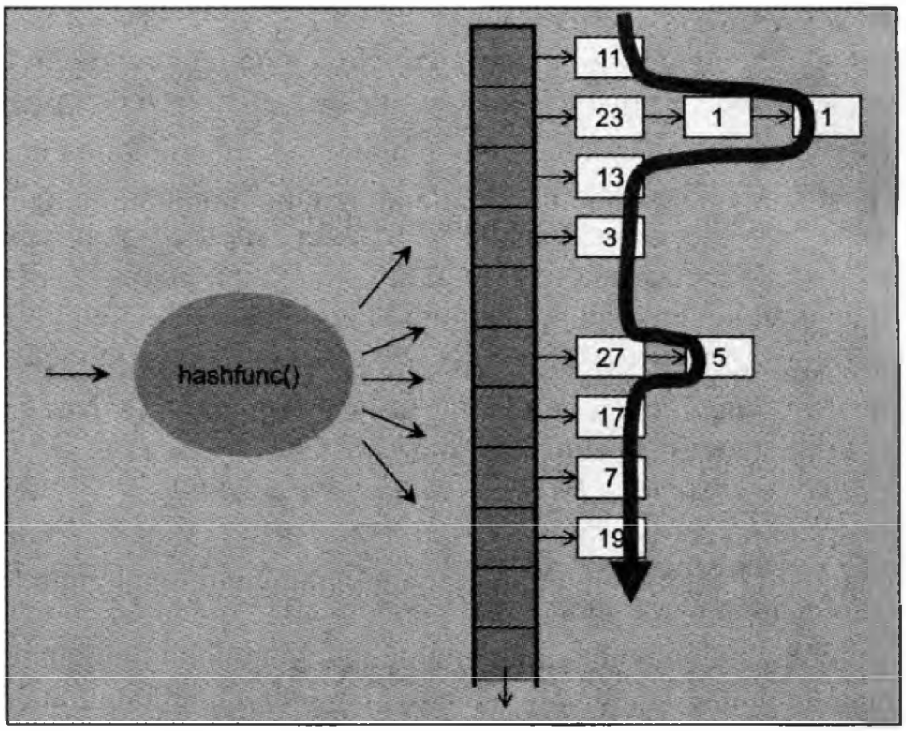


Рис. 6.8. Итератор pos перемещается по элементам неупорядоченного мультимножества

Тип внутреннего итератора `pos`, используемого в цикле `for`, предоставляется контейнером, так что итератор "знает", как обходить все элементы. При указанном порядке следования элементов внутреннее состояние неупорядоченного мультимножества может быть таким, как показано на рис. 6.8, который соответствует первому выводу всех элементов с помощью итератора.

Если переключиться на неупорядоченное множество и тем самым запретить дубликаты

```
std::unordered_set<int> coll;
```

то результат работы программы может стать таким:

```
11 23 1 13 3 27 5 17 7 19
11 23 1 13 25 3 27 5 17 7 19
```

6.3.2. Категории итераторов

Кроме основных операций, итераторы могут иметь функциональные возможности, зависящие от внутренней структуры контейнера. Как обычно, библиотека STL предусматривает лишь те операции, которые обеспечивают высокую производительность. Например, если контейнеры предоставляют произвольный доступ (например, векторы и деки), их итераторы также могут выполнять операции произвольного доступа, такие как установка итератора на пятый элемент.

Итераторы подразделяются на *категории* по их общим возможностям. Итераторы стандартных контейнерных классов относятся к одной из трех категорий.

1. **Однонаправленный итератор** может перемещаться только вперед, используя операцию инкремента. Итераторы класса `forward_list` являются прямыми.

Итераторы контейнерных классов `unordered_set`, `unordered_multiset`, `unordered_map` и `unordered_multimap` являются "по крайней мере" прямыми (библиотекам разрешается заменять их двунаправленными итераторами, описанными в разделе 7.9.1).

2. **Двунаправленные итераторы** могут обходить контейнеры в двух направлениях: вперед с помощью операции инкремента и назад с помощью операции декремента. Итераторы контейнерных классов `list`, `set`, `multiset`, `map` и `multimap` являются двунаправленными.
3. **Итераторы произвольного доступа** обладают всеми свойствами двунаправленных итераторов. Кроме того, они могут обеспечить произвольный доступ. В частности, они выполняют операции *арифметики итераторов* (по аналогии с арифметикой обычных указателей). Можно складывать и вычитать смещения, вычислять разности и сравнивать итераторы с помощью операций сравнения, например < и >. Итераторы контейнерных классов `vector`, `deque`, `array` и `string` являются итераторами произвольного доступа.

Кроме того, существуют еще две категории итераторов.

- **Итераторы ввода** могут считывать и обрабатывать значения, перемещаясь вперед. К этой категории, в частности, относятся потоковые итераторы (см. раздел 6.5.2).
- **Итераторы вывода** могут записывать значения, перемещаясь вперед. Примерами таких итераторов являются итераторы вставки (см. раздел 6.5.1) и потоковые итераторы вывода (см. раздел 6.5.2).

Подробное обсуждение всех категорий итераторов приведено в разделе 9.2.

Для создания обобщенного кода, как можно менее зависящего от типа контейнера, не следует использовать специальные операции для итераторов произвольного доступа. Например, следующий цикл будет работать со всеми контейнерами:

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    ...
}
```

Однако следующий код *не работает* со всеми контейнерами:

```
for (auto pos = coll.begin(); pos < coll.end(); ++pos) {
    ...
}
```

Единственная разница между ними заключается в использовании операции < вместо операции != в условии выхода из цикла. Операция < предусмотрена только для итераторов произвольного доступа, поэтому этот цикл не будет работать со списками, множествами и отображениями. Для того чтобы написать обобщенный код для произвольных контейнеров, следует использовать операцию !=, а не <. Однако в этом случае код становится менее безопасным, потому что можно не распознать выход итератора `pos` за позицию `end()` (см. раздел 6.12, посвященный возможным ошибкам при использовании библиотеки STL). Выбор версии зависит от программиста; он может зависеть от контекста или личных предпочтений.

Для того чтобы избежать недоразумений, следует подчеркнуть, что речь идет о *категориях*, а не о *классах* итераторов. Категория определяет лишь возможности итераторов.

Тип не имеет значения. Обобщенная концепция библиотеки STL работает с *чистой абстракцией*: все, что *ведет* себя как двунаправленный итератор, *является* двунаправленным итератором.

6.4. Алгоритмы

В библиотеке STL предусмотрено несколько стандартных алгоритмов для обработки элементов коллекций. Эти алгоритмы обеспечивают основные операции, такие как поиск, сортировка, копирование, переупорядочение, модификация и численные расчеты.

Алгоритмы не являются членами контейнерных классов. Они представляют собой глобальные функции, работающие с итераторами. Это обстоятельство обеспечивает важное преимущество: можно разрабатывать алгоритмы для всех контейнерных типов сразу, а не для каждого в отдельности. Алгоритм может даже работать с элементами контейнеров разных типов. Кроме того, можно использовать алгоритмы для контейнерных типов, определенных пользователем. В результате эта концепция позволяет уменьшить размер кода и увеличить мощь и гибкость библиотеки.

Отметим, что эта концепция относится не к объектно-ориентированной, а к парадигме функционального программирования. Вместо объединения данных и операций, как это принято в объектно-ориентированном программировании, они разделяются на отдельные части, взаимодействующие с помощью определенного интерфейса. Впрочем, эта концепция имеет и недостатки: во-первых, ее использование не является интуитивным, во-вторых, некоторые сочетания структур данных и алгоритмов могут оказаться неработоспособными. Возможна еще более плохая ситуация, когда некое сочетание контейнерного типа и алгоритма может оказаться возможным, но вредным (например, снижать быстродействие программы). Таким образом, для того чтобы извлечь максимальную пользу, необходимо хорошо изучить библиотеку STL со всеми ее достоинствами и недостатками. В оставшейся части главы приводятся примеры и подробности, относящиеся к алгоритмам.

Начнем с простого примера использования алгоритмов библиотеки STL.

```
// stl/algo1.cpp

#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
   // создаем вектор элементов от 1 до 6 в произвольном порядке
   vector<int> coll = { 2, 5, 4, 1, 6, 3 };
   // находим и выводим на экран минимальный и максимальный элементы
   auto minpos = min_element(coll.cbegin(),coll.cend());
   cout << "min: " << *minpos << endl;
   auto maxpos = max_element(coll.cbegin(),coll.cend());
   cout << "max: " << *maxpos << endl;

   // сортируем все элементы
   sort (coll.begin(), coll.end());
```

```
   // находим первый элемент со значением 3
   // - не используем cbegin()/cend(), потому что позднее мы
   // модифицируем элементы, на которые будет ссылаться pos3
   auto pos3 = find (coll.begin(), coll.end(), // диапазон
                     3);                       // значение

   // изменяем на обратный порядок элементов, начиная с равного 3
   // и до конца
   reverse (pos3, coll.end());

   // выводим на экран все элементы
   for (auto elem : coll) {
      cout << elem << ' ';
   }
   cout << endl;
}
```

Чтобы иметь возможность вызывать алгоритмы, в программу следует включить заголовочный файл <algorithm> (для некоторых алгоритмов необходимы специальные заголовочные файлы, см. раздел 11.1).

```
#include <algorithm>
```

Первые два алгоритма, `min_element()` и `max_element()`, вызываются с двумя параметрами, определяющими диапазон обрабатываемых элементов. Для обработки всех элементов контейнера используются функции `cbegin()` и `cend()` или `begin()` и `end()` соответственно. Оба алгоритма возвращают итератор, ссылающийся на позицию первого найденного элемента. Таким образом, в операторе

```
auto minpos = min_element(coll.cbegin(),coll.cend());
```

алгоритм `min_element()` возвращает позицию минимального элемента. (Если минимальных элементов несколько, алгоритм возвращает первый из них.) Следующий оператор выводит на экран элемент, на который ссылается итератор:

```
cout << "min: " << *minpos << endl;
```

Разумеется, эти операции можно совместить[8]:

```
cout << *min_element(coll.cbegin(),coll.cend()) << endl;
```

Следующий алгоритм, `sort()`, как следует из его названия, сортирует диапазон, определенный двумя аргументами. Как обычно, ему можно передать необязательный критерий сортировки. По умолчанию в качестве критерия сортировки используется операция <. Таким образом, в данном примере все элементы контейнера сортируются в возрастающем порядке.

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

[8] Здесь есть одна проблема: если контейнер `coll` пуст, будет предпринята некорректная попытка разыменования несуществующего элемента. Поэтому крайне желательна проверка возвращаемого алгоритмом значения перед его использованием. — *Примеч. консульт.*

В результате вектор содержит элементы, следующие в указанном ниже порядке.

```
1 2 3 4 5 6
```

Обратите внимание на то, что здесь мы не можем использовать функции `cbegin()` и `cend()`, потому что алгоритм `sort()` изменяет значения элементов, что невозможно для итераторов типа `const_iterator`.

Алгоритм `find()` ищет значение в заданном диапазоне. В нашем примере этот алгоритм ищет первый элемент, равный `3` во всем контейнере.

```
auto pos3 = find (coll.begin(), coll.end(), // диапазон
                  3);                       // значение
```

Если алгоритм `find()` успешно выполнен, он возвращает позицию итератора, установленного на найденный элемент. Если поиск завершился неудачно, алгоритм возвращает конец диапазона, переданный как второй аргумент. В нашем случае это запредельный итератор контейнера `coll`. Значение `3` найдено как третий элемент, поэтому итератор `pos3` ссылается на третий элемент контейнера `coll`.

Последний алгоритм — `reverse()`, вызванный в примере, изменяет на обратный порядок следования элементов в заданном диапазоне. В данном случае в качестве аргументов в алгоритм `find()` передается итератор, установленный на третий элемент, и запредельный итератор:

```
reverse (pos3, coll.end());
```

Этот вызов изменяет на обратный порядок следования элементов, расположенных в диапазоне от третьего до последнего. Поскольку эта операция является модификацией, мы должны использовать неконстантный итератор. Именно поэтому мы вызвали алгоритм `find()` с функциями `begin()` и `end()`, а не `cbegin()` и `cend()`. В противном случае итератор `pos3` был бы константным, и это привело бы к ошибке при его передаче алгоритму `reverse()`.

Результаты работы программы выглядят следующим образом:

```
min: 1
max: 6
1 2 6 5 4 3
```

Отметим, что в этом примере использовано несколько средств из стандарта C++11. Если ваша платформа не поддерживает стандарт C++11, программа может выглядеть следующим образом:

```
// stl/algo1old.cpp

#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
   // создаем вектор элементов от 1 до 6 в произвольном порядке
   vector<int> coll;
```

```
    coll.push_back(2);
    coll.push_back(5);
    coll.push_back(4);
    coll.push_back(1);
    coll.push_back(6);
    coll.push_back(3);

    // находим и выводим на экран минимальный и максимальный элементы
    vector<int>::const_iterator minpos = min_element(coll.begin(),
                                                     coll.end());
    cout << "min: " << *minpos << endl;

    vector<int>::const_iterator maxpos = max_element(coll.begin(),
                                                     coll.end());
    cout << "max: " << *maxpos << endl;

    // сортируем все элементы
    sort (coll.begin(), coll.end());

    // находим первый элемент, равный 3
    vector<int>::iterator pos3;
    pos3 = find (coll.begin(), coll.end(), // диапазон
                 3);                       // значение

    // изменяем на обратный порядок элементов, начиная с равного 3
    // и до конца
    reverse (pos3, coll.end());

    // выводим на экран все элементы
    vector<int>::const_iterator pos;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
         cout << *pos << ' ';
    }
    cout << endl;
}
```

Отличия заключаются в следующем.

- Для инициализации вектора невозможно использовать список инициализации.
- Функции-члены `cbegin()` и `cend()` не предусмотрены, поэтому вместо них приходится использовать функции `begin()` и `end()`. Тем не менее можно использовать константные итераторы.
- Вместо ключевого слова `auto` необходимо всегда явно объявлять итераторы.
- Вместо диапазонных циклов `for` для вывода на экран элементов коллекции необходимо использовать итераторы.

6.4.1. Диапазоны

Все алгоритмы обрабатывают один или несколько *диапазонов*. Эти диапазоны могут, но не обязаны содержать все элементы контейнера. Следовательно, для того чтобы

обрабатывать подмножество элементов контейнера, необходимо передавать начало и конец диапазона в качестве двух разных аргументов, а не всю коллекцию как один аргумент.

Этот интерфейс является гибким и в то же время опасным. Вызывающая функция обязана гарантировать, что первый и второй аргументы определяют *корректный* диапазон. Диапазон считается корректным, если конец диапазона *достижим* из его начала путем перебора элементов. Следовательно, программист должен самостоятельно гарантировать, что оба итератора принадлежат одному и тому же контейнеру и что начало диапазона предшествует его концу. В противном случае последствия будут непредсказуемыми, в частности, могут возникнуть бесконечные циклы и обращение к запретным областям памяти. В этом отношении итераторы так же опасны, как обычные указатели. Однако неопределенное поведение также означает, что реализация библиотеки STL может свободно распознавать такие виды ошибок и соответствующим образом обрабатывать их. Ниже будет показано, что гарантировать корректность *диапазонов* не так просто, как кажется. Более подробно ловушки и безопасные версии компонентов библиотеки STL описаны в разделе 6.12.

Каждый алгоритм обрабатывает *полуоткрытый* диапазон. Таким образом, диапазон определен так, что он содержит начало, но не содержит конец. Эту концепцию часто описывают с помощью традиционных математических обозначений:

```
[begin,end)
```

или

```
[begin,end[
```

В книге используется первый вид обозначений.

Преимущество концепции полуоткрытого диапазона заключается в том, что она проста и предотвращает работу с пустыми коллекциями (см. раздел 6.3). Однако у нее есть и недостатки. Рассмотрим следующий пример:

```
// stl/find1.cpp

#include <algorithm>
#include <list>
#include <iostream>
using namespace std;

int main()
{
   list<int> coll;

   // вставляем элементы от 20 до 40
   for (int i=20; i<=40; ++i) {
        coll.push_back(i);
   }

   // находим позицию элемента, равного 3
   // - его здесь нет, поэтому итератор pos3 равен coll.end()
   auto pos3 = find (coll.begin(), coll.end(), // диапазон
                     3);                       // значение

   // изменяем на обратный порядок следования элементов
   // от найденного до конца контейнера
   // - поскольку pos3 равен coll.end(), обратный порядок
```

```
    // устанавливается в пустом контейнере
    reverse (pos3, coll.end());

    // находим позиции значений 25 и 35
    list<int>::iterator pos25, pos35;
    pos25 = find (coll.begin(), coll.end(), // диапазон
                  25);                      // значение
    pos35 = find (coll.begin(), coll.end(), // диапазон
                  35);                      // значение

    // выводим на экран максимальный элемент заданного диапазона
    // - примечание: включая pos25, но исключая pos35
    cout << "max: " << *max_element (pos25, pos35) << endl;

    // обрабатываем элементы, включая последнюю позицию
     cout << "max: " << *max_element (pos25, ++pos35) << endl;
}
```

В этом примере коллекция инициализируется целыми числами от `20` до `40`. Когда поиск элемента, равного `3`, заканчивается неудачей, алгоритм `find()` возвращает конец обработанного диапазона (в нашем примере — `coll.end()`) и присваиваем его итератору `pos3`. Использование этого значения в качестве начала диапазона в следующем вызове алгоритма `reverse()` не создает никаких проблем, потому что он эквивалентен следующему вызову:

```
reverse (coll.end(), coll.end());
```

Это просто обращение порядка элементов пустого диапазона. Таким образом, эта операция ничего не делает (так называемая "пустая операция").

Однако если алгоритм `find()` используется для поиска первого и последнего элементов подмножества, необходимо передать эти итераторы в виде диапазона, не содержащего последний элемент. Итак, первый вызов

```
max_element()
max_element (pos25, pos35)
```

найдет число `34`, но не `35`:

```
max: 34
```

Для работы с последним элементом необходимо передать позицию, следующую за последним элементом:

```
max_element (pos25, ++pos35)
```

Это приведет в правильному результату:

```
max: 35
```

Отметим, что в этом примере в качестве контейнера используется список. Таким образом, для того чтобы получить позицию, следующую за итератором `pos35`, необходимо использовать операцию `++`. При работе с итератором произвольного доступа, например итераторами вектора или дека, можно также использовать выражение `pos35 + 1`, поскольку итераторы произвольного доступа допускают *арифметику итераторов* (см. разделы 6.3.2 и 9.2.5).

Разумеется, для поиска элементов в подынтервале можно использовать итераторы pos25 и pos35. И снова, чтобы включить pos35 в диапазон поиска, необходимо передать алгоритму позицию, следующую за этим итератором. Рассмотрим пример:

```
// увеличиваем pos35 для поиска с учетом его значения
++pos35;
pos30 = find(pos25,pos35, // диапазон
             30);         // значение
if (pos30 == pos35) {
    cout << "30 НЕ принадлежит подынтервалу" << endl;
}
else {
    cout << "30 принадлежит подынтервалу" << endl;
}
```

Все примеры в этом разделе работают только потому, что нам известно, что итератор pos25 предшествует итератору pos35. В противном случае диапазон [pos25,pos35) не был бы корректным. Если заранее неизвестно, какой элемент предшествует другому, ситуация усложняется и может возникнуть неопределенное поведение.

Допустим, что нам неизвестно, предшествует ли элемент, равный 25, элементу, равному 35. Может даже оказаться, что одно или оба этих значения в коллекции отсутствуют. Используя итератор произвольного доступа, для проверки этого условия можно вызвать операцию <.

```
if (pos25 < pos35) {
   // только [pos25,pos35) является корректным диапазоном
   ...
}
else if (pos35 < pos25) {
   // только [pos35,pos25) является корректным диапазоном
   ...
}
else {
   // оба итератора равны друг другу, значит оба должны быть равны end()
   ...
}
```

Однако при работе с итераторами, не являющимися итераторами произвольного доступа, не существует простого и быстрого способа обнаружения предшествующего итератора. Можно лишь выполнять поиск первого итератора в диапазоне от начала контейнера до второго итератора или в диапазоне от второго итератора до конца контейнера. В этом случае алгоритм можно изменить следующим образом: вместо поиска обоих значений во всем контейнере следует попробовать выяснить, какое значение оказывается первым:

```
pos25 = find (coll.begin(), coll.end(), // диапазон
              25);                      // значение
pos35 = find (coll.begin(), pos25,      // диапазон
              35);                      // значение
if (pos25 != coll.end() && pos35 != pos25) {
// итератор pos35 расположен перед итератором pos25
// поэтому корректным является только диапазон [pos35,pos25)
...
```

```
}
else {
   pos35 = find (pos25, coll.end(), // диапазон
                 35);               // значение
   if (pos35 != coll.end()) {
       // итератор pos25 предшествует итератору pos35
       // поэтому корректным является только диапазон [pos25,pos35)
       ...
   }
   else {
       // 25 и/или 35 не найдены
       ...
   }
}
```

В противоположность предыдущей версии мы не ищем 35 во всем контейнере coll. Вместо этого мы сначала ищем его в диапазоне от начала до итератора pos25. Затем, если он не найден, ищем его среди элементов, следующих за итератором pos25. В результате нам известно, какая из позиций итератора является первой и какой диапазон считается корректным.

Эта реализация не очень эффективна. Более эффективный способ найти первый элемент, равный 25 или 35, — искать их непосредственно. Это можно сделать с помощью алгоритма find_if() или лямбда-выражения (см. раздел 3.1.10), определив критерий, который применяется к каждому элементу контейнера coll.

```
pos = find_if (coll.begin(), coll.end(), // диапазон
               [] (int i) {              // критерий
                   return i == 25 || i == 35;
               });
if (pos == coll.end()) {
   // элемент, равный 25 или 35, не обнаружен
   ...
}
else if (*pos == 25) {
   // элемент, равный 25, является первым
   pos25 = pos;
   pos35 = find (++pos, coll.end(), // диапазон
                 35);               // значение
   ...
}
else {
   // элемент, равный 35, является первым
   pos35 = pos;
   pos25 = find (++pos, coll.end(), // диапазон
                 25);               // значение
   ...
}
```

Здесь специальное лямбда-выражение

```
[] (int i) {
return i == 25 || i == 35;
}
```

используется в качестве критерия, позволяющего выполнять поиск первого элемента, равного 25 или 35. Использование лямбда-выражений в библиотеке STL описано в разделе 6.9 и обсуждается в разделе 10.3.

6.4.2. Обработка нескольких диапазонов

Некоторые алгоритмы работают с несколькими диапазонами. В этом случае обычно необходимо определить начало и конец только первого диапазона. Для всех остальных *диапазонов* необходимо передать лишь их начало. Концы остальных *диапазонов* определяются количеством элементов в первом диапазоне. Например, следующий вызов алгоритма `equal()` сравнивает все элементы коллекции `coll1` с элементами коллекции `coll2`, начиная с первого элемента:

```
if (equal (coll1.begin(), coll1.end(), // первый диапазон
           coll2.begin())) {           // второй диапазон
   ...
}
```

Таким образом, количество элементов коллекции `coll2`, сравниваемых с элементами коллекции `coll1`, задается косвенно с помощью количества элементов коллекции `coll1` (рис. 6.9).

Это приводит к важному последствию: **при вызове алгоритмов для нескольких диапазонов второй и последующие диапазоны должны иметь не меньше элементов, чем первый диапазон**. В частности, диапазоны назначения должны быть достаточно большими, чтобы алгоритмы могли выполнить запись.

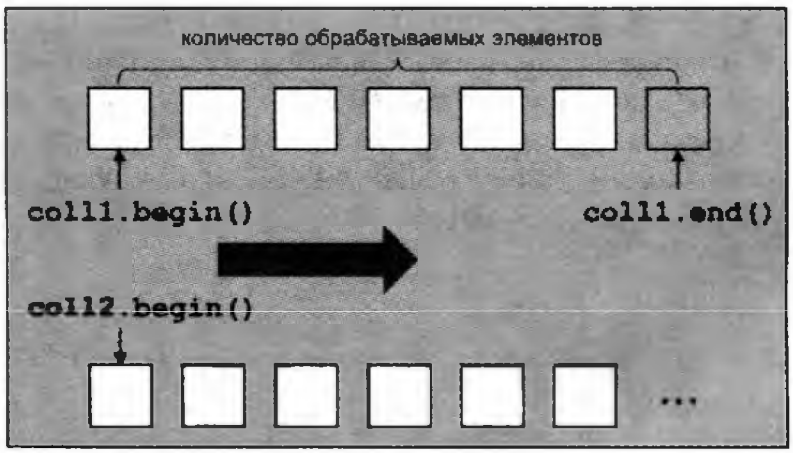


Рис. 6.9. Алгоритм, выполняющий обход двух диапазонов

Рассмотрим следующую программу:

```
// stl/copybug.cpp

#include <algorithm>
#include <list>
#include <vector>
using namespace std;

int main()
```

```
{
   list<int> coll1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
   vector<int> coll2;

   // ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ПРОГРАММЫ:
   // - перезапись несуществующих элементов в диапазоне назначения
   copy (coll1.cbegin(), coll1.cend(), // источник
         coll2.begin());               // назначение
   ...
}
```

Здесь вызывается алгоритм `copy()`, который просто копирует все элементы первого диапазона в диапазон назначения. Как обычно, определяются начало и конец первого диапазона и лишь начало второго. Однако алгоритм выполняет перезапись, а не вставку. По этой причине алгоритм *требует*, чтобы в диапазоне назначения содержалось достаточное количество элементов для перезаписи. Если места недостаточно, как в данном случае, возникает неопределенное поведение. На практике это часто означает, что происходит перезапись информации, расположенной за итератором `coll2.end()`. Если повезет, дело ограничится сбоем программы, и вы поймете, что сделали что-то не так. Впрочем, можно подстраховаться с помощью безопасной версии библиотеки STL, в которой неопределенное поведение приводит к вызову процедуры обработки ошибок (см. раздел 6.12.1).

Для того чтобы избежать этих ошибок, можно 1) сделать так, чтобы диапазон назначения имел достаточно места для записи, или 2) использовать *итераторы вставки*. Эти итераторы описаны в разделе 6.5.1. Сначала посмотрим, как можно изменить диапазон назначения, чтобы он имел достаточно места для записи.

Для того чтобы диапазон назначения стал достаточно большим, можно либо сразу задать правильный размер, либо изменить его явным образом. Обе эти альтернативы применимы только к некоторым последовательным контейнерам (`vector`, `deque`, `list` и `forward_list`). Однако для остальных контейнеров это вообще не проблема, поскольку ассоциативные и неупорядоченные контейнеры нельзя использовать в качестве диапазонов назначения в алгоритмах перезаписи (см. раздел 6.7.2). Следующая программа демонстрирует способ увеличения размера контейнера:

```
// stl/copy1.cpp

#include <algorithm>
#include <list>
#include <vector>
#include <deque>
using namespace std;

int main()
{
   list<int> coll1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
   vector<int> coll2;

   // изменяем размер диапазона назначения, чтобы он имел
   // достаточно места для алгоритма перезаписи
   coll2.resize (coll1.size());

   // копируем элементы из первой коллекции во вторую
```

```
    // - перезаписываем существующие элементы в диапазон назначения
    copy (coll1.cbegin(), coll1.cend(), // источник
          coll2.begin());               // назначение

    // создаем третью коллекцию, имеющую достаточный объем
    // - начальный размер передается как параметр
    deque<int> coll3(coll1.size());

    // копируем элементы из первой в третью коллекцию
    copy (coll1.cbegin(), coll1.cend(), // источник
          coll3.begin());               // назначение
}
```

Здесь функция `resize()` используется для изменения количества элементов в существующем контейнере `coll2`.

```
coll2.resize (coll1.size());
```

Затем коллекция `coll3` инициализируется специальным начальным размером, обеспечивающим достаточный объем для хранения всех элементов коллекции `coll1`.

```
deque<int> coll3(coll1.size());
```

Отметим, что изменение и инициализация размера приводят к созданию новых элементов, которые инициализируются своим конструктором, заданным по умолчанию, поскольку им не передаются никакие аргументы. Кроме того, конструктору и функции `resize()` можно передать дополнительный аргумент для инициализации новых элементов.

6.5. Адаптеры итераторов

Итераторы являются *чистой абстракцией*: все, что *ведет себя* как итератор, *является* итератором. По этой причине можно написать классы, имеющие интерфейс итераторов, но делающие нечто совершенно другое. Стандартная библиотека C++ содержит несколько стандартных итераторов особого вида: *адаптеры итераторов*. Это нечто большее, чем просто вспомогательные классы; они придают концепции итераторов много больше мощи.

В следующих подразделах рассматриваются перечисленные ниже адаптеры итераторов.

1. Итераторы вставки.
2. Потоковые итераторы.
3. Обратные итераторы.
4. Итераторы перемещения (начиная со стандарта C++11).

Подробное описание адаптеров итераторов приведено в разделе 9.4.

6.5.1. Итераторы вставки

Итераторы вставки используются для того, чтобы дать алгоритмам возможность работать в режиме вставки, а не замены. В частности, итераторы вставки решают проблему, возникающую, когда в диапазоне назначения нет достаточного места: они позволяют увеличивать размер диапазона вставки.

Итераторы вставки переопределяют свой интерфейс следующим образом.

- Если вы присваиваете значение элементу итератора вставки, он вставляет это значение в коллекцию, которой принадлежат итератор. Три разных итератора вставки имеют разные возможности для вставки элементов — в начало, в конец и в заданную позицию.
- Вызов для перемещения вперед является пустой операцией.

Итераторы с таким интерфейсом относятся к категории итераторов вывода, способных записывать и присваивать значения только при перемещении вперед (см. раздел 9.2).

Рассмотрим пример:

```
// stl/copy2.cpp

#include <algorithm>
#include <iterator>
#include <list>
#include <vector>
#include <deque>
#include <set>
#include <iostream>
using namespace std;

int main()
{
   list<int> coll1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
   // копируем элементы коллекции coll1 в коллекцию coll2,
   // добавляя их к существующим

   vector<int> coll2;
   copy (coll1.cbegin(), coll1.cend(), // источник
         back_inserter(coll2));        // назначение

   // копируем элементы из коллекции coll1 в коллекцию coll3,
   // вставляя их в начало
   // - изменяем порядок на обратный
   deque<int> coll3;
   copy (coll1.cbegin(), coll1.cend(), // источник
         front_inserter(coll3));       // назначение

   // копируем элементы коллекции coll1 в коллекцию coll4
   // - итератор, работающий только с ассоциативными коллекциями
   set<int> coll4;
   copy (coll1.cbegin(), coll1.cend(),   // источник
         inserter(coll4,coll4.begin())); // назначение
}
```

Этот пример демонстрирует все три стандартных итератора вставки.

1. **Итераторы вставки в конец** вставляют элементы в конец своего контейнера (добавляет их), вызывая функцию `push_back()`. Например, в следующем примере все элементы коллекции `coll1` добавляются в коллекцию `coll2`:

```
copy (coll1.cbegin(), coll1.cend(), // источник
      back_inserter(coll2));        // назначение
```

2. Разумеется, итераторы вставки в конец можно использовать только для контейнеров, имеющих функцию-член `push_back()`. В стандартной библиотеке C++ к таким контейнерам относятся `vector`, `deque`, `list` и строки.
3. **Итераторы вставки в начало** вставляют элементы в начало своего контейнера, вызывая функцию `push_front()`. Например, следующий оператор вставляет все элементы коллекции `coll1` в коллекцию `coll3`:

```
copy (coll1.cbegin(), coll1.cend(), // источник
      front_inserter(coll3));       // назначение
```

4. Отметим, что этот вид итераторов вставки изменяет порядок вставленных элементов на противоположный. Если вставить в начало `1`, а затем `2`, то `1` будет следовать за `2`.
5. Итераторы вставки в начало можно использовать только в контейнерах, имеющих функцию-член `push_front()`. В стандартной библиотеке C++ к таким контейнерам относятся `deque`, `list` и `forward_list`.
6. **Обобщенные итераторы вставки** вставляют элементы непосредственно перед позицией, передаваемой как второй аргумент во время инициализации. Обобщенный итератор вставки вызывает функцию-член `insert()` с новым значением и новой позицией, задаваемыми аргументами. Отметим, что функцию-член `insert()` содержат все стандартные контейнеры, за исключением `array` и `forward_list`. Таким образом, этот итератор является единственным итератором для работы со стандартными ассоциативными и неупорядоченными контейнерами.
7. Однако постойте! Передавать позицию для вставки элемента в ассоциативный или неупорядоченный массив не слишком полезно, не так ли? В ассоциативных контейнерах позиции зависят от значений *элементов*, а в неупорядоченных контейнерах позиция элемента не определена. Решение простое: для ассоциативных и неупорядоченных контейнеров передаваемая позиция считается *подсказкой* для начала поиска правильной позиции. Однако контейнер может ее игнорировать. В разделе 9.6 описываются итераторы вставки, определенные пользователем, которые могут оказаться более полезными для работы с ассоциативными и неупорядоченными контейнерами.

Функциональные возможности итераторов вставки перечислены в табл. 6.1. Дополнительные подробности приведены в разделе 9.4.2.

Таблица 6.1. Стандартные итераторы вставки

Выражение	Вид итератора вставки
`back_inserter(`*контейнер*`)`	Добавляет элементы в том же порядке, используя функцию `push_back(`*значение*`)`
`front_inserter(`*контейнер*`)`	Вставляет элемент в начало в обратном порядке, используя функцию `push_front(`*значение*`)`
`inserter(`*контейнер*`,` *позиция*`)`	Вставляет элементы в указанную *позицию* (в том же порядке), используя функцию `insert(`*позиция*`,` *значение*`)`

6.5.2. Потоковые итераторы

Потоковые итераторы считывают данные из потока или записывают данные в поток[9]. Таким образом, они создают абстракцию, позволяющую вводить данные с клавиатуры, которая рассматривается как коллекция, предназначенная для чтения. Аналогично с помощью потокового итератора можно перенаправить вывод алгоритма непосредственно в файл или на экран.

Следующий пример демонстрирует мощь всей библиотеки STL. По сравнению с обычными программами на языке C или C++, этот пример выполняет очень сложную работу с помощью нескольких операций.

```
// stl/ioiter1.cpp


#include <iterator>
#include <algorithm>
#include <vector>
#include <string>
#include <iostream>
using namespace std;

int main()
{
   vector<string> coll;
   // считываем все слова из стандартного потока ввода
   // - источник: все строки вплоть до конца файла (или ошибки)
   // - назначение: coll (вставка)
   copy (istream_iterator<string>(cin), // начало источника
         istream_iterator<string>(),    // конец источника
         back_inserter(coll));          // назначение

   // сортируем элементы
   sort (coll.begin(), coll.end());

   // выводим все элементы без дубликатов
   // - источник: coll
   // - назначение: стандартный поток вывода
   // (с символом перехода на новую строку между элементами)
   unique_copy (coll.cbegin(), coll.cend(),           // источник
                ostream_iterator<string>(cout,"\n")); // назначение
}
```

Эта программа содержит только три оператора, которые считывают все слова из стандартного потока ввода и выводят их упорядоченный список. Рассмотрим эти три оператора по очереди. В операторе

```
copy (istream_iterator<string>(cin),
      istream_iterator<string>(),
      back_inserter(coll));
```

используются два потоковых итератора ввода.

[9] Поток — это объект, представляющий каналы ввода–вывода (см. главу 15).

1. Выражение

   ```
   istream_iterator<string>(cin)
   ```

2. создает потоковый итератор, считывающий данные из стандартного потока ввода `cin`. Шаблонный аргумент `string` указывает, что потоковый итератор считывает элементы данного типа (строки описаны в главе 13). Эти элементы считываются с помощью обычной операции ввода >>. Таким образом, каждый раз, когда алгоритм хочет обработать новый элемент, потоковый итератор ввода трансформирует это желание в вызов `cin >> строка`.

3. Операция ввода строк обычно считывает одно слово, отделенное разделителями (см. раздел 13.2.10), так что алгоритм считывает слово за словом.

4. Выражение

   ```
   istream_iterator<string>()
   ```

5. вызывает конструктор потоковых итераторов, заданный по умолчанию, который создает так называемый *итератор конца потока* (end-of-stream iterator). Он представляет поток, чтение из которого больше невозможно.

Как обычно, алгоритм `copy()` работает, только если первый аргумент (инкрементированный) отличается от второго. Итератор конца потока используется как *конец диапазона* (end of the range), поэтому алгоритм считывает все строки из потока `cin`, пока может (т.е. пока не достигнет конца потока или не произойдет ошибка). Подводя итоги, отметим, что источником алгоритма являются "все слова, считанные из потока `cin`". Эти слова копируются путем вставки в коллекцию `coll` с помощью обратного итератора.

Алгоритм `sort()` сортирует все элементы:

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

И наконец, оператор

```
unique_copy (coll.cbegin(), coll.cend(),
ostream_iterator<string>(cout,"\n"));
```

копирует все элементы из коллекции в выходной поток `cout`. На протяжении этого процесса алгоритм `unique_copy()` удаляет дубликаты. Выражение

```
ostream_iterator<string>(cout,"\n")
```

создает итератор потока вывода, записывающий объекты класса `string` в поток `cout`, выполняя операцию << для каждого элемента.

Второй аргумент, следующий за `cout`, является необязательным и служит в качестве разделителя между элементами. В нашем примере этим разделителем является символ перехода на новую строку, поэтому каждый элемент будет записан в отдельной строке.

Все компоненты нашей программы являются шаблонными, поэтому ее легко настроить на сортировку любых других типов, таких как целые числа или более сложные объекты. Более подробное описание и примеры использования итераторов ввода-вывода приведены в разделе 9.4.3.

В нашем примере для сортировки всех слов, введенных из стандартного потока ввода, использовалось одно объявление и три инструкции. Однако то же самое можно сделать

с помощью одного объявления и одной инструкции. Соответствующий пример приведен в разделе 1.

Начиная со стандарта C++11 в качестве параметра, задающего конца диапазона можно передавать пустые фигурные скобки, а не итератор потока, созданный по умолчанию. Это возможно благодаря тому, что тип аргумента, определяющего конец диапазона, выводится из предыдущего аргумента, определяющего начало диапазона:

```
copy (istream_iterator<string>(cin), // начало диапазона-источника
      {},                            // конец диапазона-источника
      back_inserter(coll));          // диапазон-получатель
```

6.5.3. Обратные итераторы

Обратные итераторы позволяют алгоритмам перебирать элементы контейнера в обратном направлении, заменяя операцию инкремента на операцию декремента, и наоборот. Все контейнеры с двунаправленными итераторами или операциями произвольного доступа (т.е. все последовательные контейнеры, за исключением `forward_list` и всех ассоциативных контейнеров) могут создавать обратные итераторы с помощью функций-членов `rbegin()` и `rend()`. В соответствии со стандартом C++11 в контейнерах также предусмотрены функции-члены, возвращающие итераторы, предназначенные только для чтения, `crbegin()` и `crend()`.

В контейнерах `forward_list` и неупорядоченных контейнерах интерфейс для обратного обхода (`rbegin()`, `rend()` и т.д.) не предусмотрен. Причина заключается в том, что их реализация для обхода элементов использует только односвязные списки.

Рассмотрим следующий пример:

```
// stl/reviter1.cpp

#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
   vector<int> coll;

   // вставляем элементы от 1 до 9
   for (int i=1; i<=9; ++i) {
        coll.push_back(i);
   }

   // выводим на экран все элементы в обратном порядке
   copy (coll.crbegin(), coll.crend(),       // источник
         ostream_iterator<int>(cout," "));   // назначение
   cout << endl;
}
```

Следующее выражение возвращает обратный итератор, предназначенный только для чтения элементов контейнера `coll`.

```
coll.crbegin()
```

Этот итератор можно использовать в качестве начала обратного обхода элементов коллекции. Он установлен на последний элемент коллекции. Таким образом, следующее выражение возвращает значение последнего элемента:

```
*coll.crbegin()
```

Соответственно, следующее выражение возвращает обратный итератор для коллекции `coll`, который можно использовать в качестве конца для обратного обхода:

```
coll.crend()
```

Как обычно при работе с диапазонами, позиция итератора следует за последним элементом, но в обратном направлении; иначе говоря, она расположена *перед* первым элементом коллекции.

Никогда не следует применять операцию `*` (или операцию `->`) к позиции, не содержащей корректного элемента. Таким образом, выражение

```
*coll.crend()
```

не определено, так же как и выражения `*coll.end()` и `*coll.cend()`.

Преимущество использования обратных итераторов заключается в том, что все алгоритмы способны обходить элементы коллекции в обратном направлении без специального кода. Переход к следующему элементу с помощью операции `++` переопределяется с помощью операции `--`. Например, в нашем случае алгоритм `copy()` обходит элементы коллекции `coll` от последнего к первому элементу. Таким образом, получаем следующие результаты работы программы:

```
9 8 7 6 5 4 3 2 1
```

Обычные итераторы также можно превращать в обратные, и наоборот. Однако разыменованное значение итератора при этом изменяется. Подробности, касающиеся обратных итераторов, изложены в разделе 9.4.1.

6.5.4. Итераторы перемещения

Итераторы перемещения введены в стандарте C++11. Они превращают любой доступ к элементу в операцию перемещения. Это позволяет перемещать элементы из одного контейнера в другой либо в конструкторах, либо с помощью алгоритмов. Подробности изложены в разделе 9.4.4.

6.6. Пользовательские обобщенные функции

Библиотека STL допускает расширение. Это означает, что пользователь может писать свои функции и алгоритмы для работы с элементами коллекции. Разумеется, эти операции могут быть обобщенными. Однако для объявления корректного итератора в этих операциях необходимо использовать тип контейнера, который у каждого контейнера свой. Для того чтобы облегчить создание обобщенных функций, каждый контейнерный тип предусматривает внутренние определения типов. Рассмотрим следующим пример:

```
// stl/print.hpp

#include <iostream>
#include <string>

// PRINT_ELEMENTS()
// - выводит необязательную строку optstr, за которой следуют
// - все элементы коллекции coll
// - в одной строке, разделенные пробелами
template <typename T>
inline void PRINT_ELEMENTS (const T& coll,
                            const std::string& optstr="")
{
   std::cout << optstr;
   for (const auto& elem : coll) {
        std::cout << elem << ' ';
   }
   std::cout << std::endl;
}
```

В этом примере определена обобщенная функция, которая выводит на экран необязательную строку, за которой следуют все элементы переданного контейнера.

До появления стандарта C++11 цикл по элементам выглядел следующим образом:

```
typename T::const_iterator pos;
for (pos=coll.begin(); pos!=coll.end(); ++pos) {
     std::cout << *pos << ' ';
}
```

где переменная `pos` объявлена как итератор передаваемого контейнера. Отметим, что ключевое слово `typename` здесь является обязательным, чтобы указать, что `const_iterator` — это тип, а не статический член типа `T` (см. раздел 3.2).

В дополнение к типам `iterator` и `const_iterator` контейнеры содержат другие типы для создания обобщенных функций. Например, они предоставляют тип элементов для создания временных копий элементов. Подробности изложены в разделе 9.5.1.

Необязательный второй аргумент `PRINT_ELEMENTS` представляет собой строку, используемую как префикс перед записываемыми элементами. Таким образом, функция `PRINT_ELEMENTS()` позволяет комментировать результаты работы программы:

```
PRINT_ELEMENTS (coll, "все элементы: ");
```

Эта функция описана здесь потому, что она будет широко использоваться в остальной части книги при выводе элементов контейнера.

6.7. Модифицирующие алгоритмы

До сих пор мы рассматривали концепцию библиотеки STL в целом. Контейнеры представляют разные способы управления коллекциями данных, алгоритмы выполняют операции чтения и записи элементов этих коллекций. Итераторы служат посредниками между контейнерами и алгоритмами. Итераторы, предоставляемые контейнерами, позволяют перебирать все элементы в разном порядке и в разных режимах, например в режиме вставки.

Однако настало время сказать "однако". На практике работа с библиотекой STL связана с определенными ограничениями, которые следует знать. Многие из них относятся к модификациям.

Несколько алгоритмов модифицируют целевые диапазоны. В частности, эти алгоритмы могут удалять элементы. При этом возникают определенные аспекты, о которых речь пойдет в этом разделе. Эти аспекты вызывают удивление и демонстрируют обратную сторону концепции STL, которая подразумевает разделение контейнеров и алгоритмов для достижения высокой гибкости.

6.7.1. Удаление элементов

Алгоритм `remove()` удаляет элементы из диапазона. Однако применение этого алгоритма ко всем элементам контейнера приводит к удивительным результатам. Рассмотрим следующий пример:

```
// stl/remove1.cpp

#include <algorithm>
#include <iterator>
#include <list>
#include <iostream>
using namespace std;

int main()
{
   list<int> coll;

   // вставляем элементы от 6 до 1 и от 1 до 6
   for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
   }

   // выводим на экран все элементы коллекции
   cout << "pre:  ";
   copy (coll.cbegin(), coll.cend(),       // источник
         ostream_iterator<int>(cout," ")); // назначение
   cout << endl;

   // удаляем все элементы, равные 3
   remove (coll.begin(), coll.end(), // диапазон
           3);                       // значение

   // выводим на экран все элементы коллекции
   cout << "post: ";
   copy (coll.cbegin(), coll.cend(),       // источник
         ostream_iterator<int>(cout," ")); // назначение
   cout << endl;
}
```

Человек, поверхностно разбирающийся в библиотеке STL, при чтении этой программы мог бы ожидать, что из коллекции будут удалены все элементы, равные 3. Однако результат программы выглядит следующим образом:

```
pre:  6 5 4 3 2 1 1 2 3 4 5 6
post: 6 5 4 2 1 1 2 4 5 6 5 6
```

Таким образом, алгоритм `remove()` не изменяет количество элементов в коллекции, к которой он применяется. Функция-член `cend()` возвращает старый конец — как и функция `end()`, — а функция `size()` возвращает старое количество элементов. Однако кое-что изменилось: элементы изменили порядок следования, как если бы они были удалены. Каждый элемент, равный 3, заменяется следующими элементами (см. рис. 6.10). В конце коллекции старые элементы не заменяются алгоритмом и остаются неизменными. Логически эти элементы больше не принадлежат коллекции.

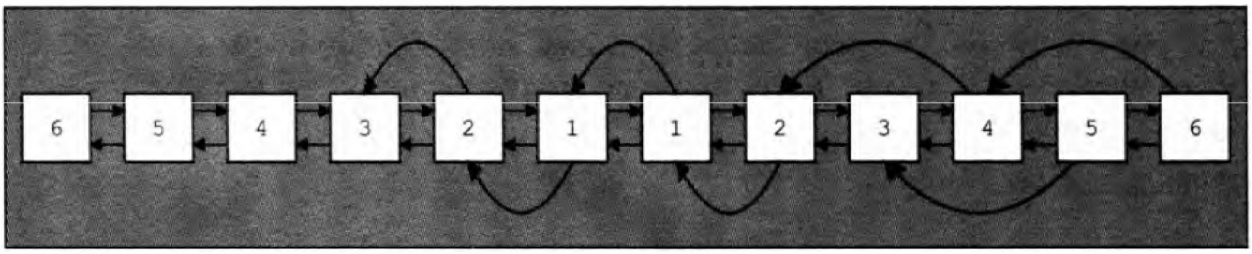


Рис. 6.10. Как работает алгоритм ***remove()***

Однако этот алгоритм возвращает новый логический конец. С помощью этого алгоритма можно получить доступ к результирующему диапазону, уменьшить размер коллекции или подсчитать количество удаленных элементов. Рассмотрим следующую модифицированную версию этого примера:

```
// stl/remove2.cpp

#include <algorithm>
#include <iterator>
#include <list>
#include <iostream>
using namespace std;

int main()
{
   list<int> coll;

   // вставляем элементы от 6 до 1 и от 1 до 6
   for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
   }

   // выводим на экран все элементы коллекции
   copy (coll.cbegin(), coll.cend(),
         ostream_iterator<int>(cout," "));
   cout << endl;

   // удаляем все элементы, равные 3
   // - сохраняем новый конец
   list<int>::iterator end = remove (coll.begin(), coll.end(),
                                     3);
```

```
    // выводим на экран результирующие элементы коллекции
    copy (coll.begin(), end,
          ostream_iterator<int>(cout," "));
    cout << endl;

    // выводим на экран количество удаленных элементов
    cout << "number of removed elements "
         << distance(end,coll.end()) << endl;

    // удаляем "удаленные" элементы
    coll.erase (end, coll.end());

    // выводим на экран все элементы модифицированной коллекции
    copy (coll.cbegin(), coll.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl;

}
```

В этой версии значение, возвращаемое алгоритмом `remove()`, присваивается итератору `end`:

```
list<int>::iterator end = remove (coll.begin(), coll.end(), 3);
```

Это новый логический конец модифицированной коллекции после "удаления" элементов. Это возвращаемое значение можно использовать в качестве нового конца для последующих операций:

```
copy (coll.begin(), end, ostream_iterator<int>(cout," "));
```

Отметим, что мы должны использовать функцию `begin()`, а не `cbegin()`, потому что итератор `end` определен как неконстантный, а начало и конец диапазона определены как итераторы одинакового типа.

Другая возможность — вычислить количество удаленных элементов, определив расстояние между "логическим" и реальным концом коллекции.

```
cout << "number of removed elements: "

<< distance(end,coll.end()) << endl;
```

Здесь используется специальная вспомогательная функция `distance()` для итераторов, которая возвращает расстояние между итераторами. Если бы итераторы были итераторами произвольного доступа, можно было бы вычислить разность непосредственно с помощью операции `-`. Однако в данном случае контейнер является списком, поэтому он предусматривает двунаправленные итераторы. Подробное описание функции `distance()` приведено в разделе 9.3.3.

Если действительно необходимо удалить "удаленные" элементы, тогда нужно вызвать соответствующую функцию-член контейнера. Для этой цели контейнеры содержат функцию-член `erase()`, удаляющую весь диапазон, заданный с помощью аргументов.

```
coll.erase (end, coll.end());
```

Результаты работы всей программы выглядят следующим образом:

```
6 5 4 3 2 1 1 2 3 4 5 6
6 5 4 2 1 1 2 4 5 6
number of removed elements: 2
6 5 4 2 1 1 2 4 5 6
```

Если необходимо удалить элементы с помощью одного оператора, можно выполнить инструкцию

```
coll.erase (remove(coll.begin(),coll.end(),3), coll.end());
```

Почему алгоритмы сами не вызывают функцию `erase()`? Этот вопрос подчеркивает цену, которую приходится платить за гибкость библиотеки STL. Библиотека STL отделяет структуры данных от алгоритмов, используя итераторы в качестве интерфейса. Однако итераторы — это абстракция позиции в контейнере. В принципе, итераторы *ничего не знают* о своих контейнерах. Таким образом, алгоритмы, использующие итераторы для доступа к элементам контейнера, не могут вызывать его функции-члены.

Эта схема имеет важные последствия, поскольку позволяет алгоритмам работать с диапазонами, которые отличаются от "всех элементов контейнера". Например, диапазон может быть подмножеством всех элементов коллекции. Он даже может быть контейнером, не имеющим функции-члена `erase()` (примером такого контейнера является массив). Итак, для того чтобы алгоритм был как можно более гибким, желательно не требовать, чтобы итератор знал о контейнере.

Отметим, что физически удалять "удаленные" элементы часто не требуется. Во многих ситуациях нет ничего страшного в том, что вместо реального конца контейнера возвращается его логический конец. В частности, все алгоритмы можно применять, указывая этот новый логический конец.

6.7.2. Работа с ассоциативными и неупорядоченными контейнерами

Модифицирующие алгоритмы, т.е. алгоритмы, удаляющие элементы, изменяющие их порядок или их значения, порождают другую проблему при их использовании для ассоциативных или неупорядоченных контейнеров: такие контейнеры нельзя использовать в качестве целевого диапазона. Причина проста: при работе с ассоциативными и неупорядоченными контейнерами модифицирующие алгоритмы могут изменять значения или позиции элементов, тем самым нарушая порядок следования элементов в контейнере (упорядоченный в ассоциативных контейнерах или определенный хеш-функцией в неупорядоченных контейнерах). Для того чтобы избежать нарушения внутреннего порядка, каждый итератор для ассоциативного и неупорядоченного контейнера объявляется как итератор с константным значением или ключом. В результате манипуляция элементами в ассоциативных или неупорядоченных контейнерах распознается на этапе компиляции как ошибка[10].

Из-за этой проблемы также нельзя выполнять алгоритмы удаления для ассоциативных контейнеров, потому что эти алгоритмы неявно манипулируют элементами. Значения удаленных элементов перезаписываются следующими элементами, которые не удаляются.

[10] К сожалению, некоторые системы плохо обрабатывают ошибки. Вы можете видеть, что нечто происходит неправильно, но не понимать почему.

Возникает вопрос: как же удалить элементы из ассоциативных контейнеров? Ответ простой: вызвать их функции-члены! Каждый ассоциативный и неупорядоченный контейнер содержит функцию-член для удаления элементов. Например, для удаления элементов можно вызвать функцию-член `erase()`.

```
// stl/remove3.cpp

#include <set>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
   // неупорядоченное множество с элементами от 1 до 9
   set<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

   // выводим на экран все элементы коллекции
   copy (coll.cbegin(), coll.cend(),
         ostream_iterator<int>(cout," "));
   cout << endl;

   // Удаляем все элементы, значение которых равно 3,
   // - алгоритм remove() не работает
   // - вместо него применяется функция-член erase()
   int num = coll.erase(3);

   // выводим на экран количество удаленных элементов
   cout << "количество удаленных элементов: " << num << endl;

   // выводим на экран все элементы модифицированной коллекции
   copy (coll.cbegin(), coll.cend(),
         ostream_iterator<int>(cout," "));
   cout << endl;
}
```

Отметим, что контейнеры содержат разные функции-члены `erase()`. Количество удаленных элементов возвращает только вариант, удаляющий элемент по значению, заданному как единственный аргумент (см. раздел 8.7.3). Разумеется, если дубликаты запрещены, то возвращаемое значение может быть равным только 0 или 1. В частности, это относится к контейнерам `set`, `map`, `unordered_set` и `unordered_map`.

Результаты работы программы выглядят следующим образом:

```
1 2 3 4 5 6 7 8 9
количество удаленных элементов: 1
1 2 4 5 6 7 8 9
```

6.7.3. Алгоритмы и функции-члены

Даже если алгоритм формально можно применить, это может оказаться неудачной идеей. Контейнер может иметь функции-члены, обеспечивающие более высокую производительность. Ярким примером является использование функции `remove()` для удаления

элементов из списка. Если применить функцию remove() к элементам списка, алгоритм не будет знать, что он работает со списком, и будет работать с ним как с обычным контейнером: переупорядочит элементы, изменив их значения. Например, если алгоритм удалит первый элемент, то все остальные элементы будут присвоены их предшественникам. Это поведение противоречит основному преимуществу списков: возможности вставлять, перемещать и удалять элементы, модифицируя связи, а не значения.

Для того чтобы не снижать производительность, списки имеют специальные функции-члены для всех модифицирующих алгоритмов. Предпочтение всегда следует отдавать функциям-членам. Более того, эти функции-члены действительно удаляют удаленные элементы, как показано в следующем примере:

```
// stl/remove4.cpp

#include <list>
#include <algorithm>
using namespace std;

int main()
{
   list<int> coll;

   // вставляем элементы от 6 до 1 и от 1 до 6
   for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
   }

   // удаляем все элементы со значением 3 (низкая производительность)
   coll.erase (remove(coll.begin(),coll.end(),
               3),
               coll.end());

   // удаляем все элементы со значением 4 (высокая производительность)
   coll.remove (4);
}
```

Если целью является высокое быстродействие программы, то всегда следует отдавать предпочтение функциям-членам, а не алгоритмам. Проблема лишь в том, чтобы знать о существовании функции-члена, обеспечивающей намного более высокую производительность для конкретного контейнера. Если применить алгоритм remove() к списку, вы не получите никакого предупреждения или сообщения об ошибке. Однако, если в этой ситуации выбрать функцию-член, то при изменении типа контейнера придется изменять весь код. В главе 11 указано, существует ли функция-член, обеспечивающая более высокую производительность, чем алгоритм.

6.8. Функции в качестве аргументов алгоритма

Для повышения гибкости и мощи программы некоторым алгоритмам можно передавать вспомогательные функции, определенные пользователем. Эти функции вызываются в алгоритмах.

6.8.1. Использование функций в качестве аргументов алгоритмов

Простейший пример — алгоритм `for_each()`, применяющий пользовательскую функцию к каждому элементу в заданном диапазоне. Рассмотрим следующий пример:

```
// stl/foreach1.cpp

#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

// функция, выводящая на экран передаваемый аргумент
void print (int elem)
{
   cout << elem << ' ';
}

int main()
{
   vector<int> coll;

   // вставляем элементы от 1 до 9
   for (int i=1; i<=9; ++i) {
        coll.push_back(i);
   }

   // выводим на экран все элементы
   for_each (coll.cbegin(), coll.cend(), // диапазон
             print);                     // операция
   cout << endl;
}
```

Алгоритм `for_each()` применяет передаваемую функцию `print()` к каждому элементу в диапазоне [`coll.cbegin()`,`coll.cend()`). Таким образом, получаем следующий результат работы программы:

```
1 2 3 4 5 6 7 8 9
```

Алгоритмы используют вспомогательные функции в разных вариантах: иногда необязательно, иногда обязательно. В частности, вспомогательные функции можно использовать для того, чтобы задать критерий поиска или сортировки или для определения операции при передаче элементов из одной коллекции в другую.

Рассмотрим еще один пример:

```
// stl/transform1.cpp

#include <set>
#include <vector>
#include <algorithm>
#include <iterator>
```

```
#include <iostream>
#include "print.hpp"

int square (int value)
{
   return value*value;
}

int main()
{
   std::set<int> coll1;
   std::vector<int> coll2;

   // вставляем элементы от 1 до 9 в коллекцию coll1
   for (int i=1; i<=9; ++i) {
        coll1.insert(i);
   }
   PRINT_ELEMENTS(coll1,"инициализированные: ");

   // переносим каждый элемент из коллекции coll1 в коллекцию coll2
   // - возводим в квадрат каждое передаваемое значение
   std::transform (coll1.cbegin(),coll1.cend(), // источник
                   std::back_inserter(coll2),   // назначение
                   square); // операция
   PRINT_ELEMENTS(coll2,"в квадрате: ");
}
```

В данном примере для возведения в квадрат каждого элемента коллекции `coll1`, передаваемого в коллекцию `colls2`, используется функция `square()` (рис. 6.11). Программа выводит на экран следующие строки:

```
инициализированные: 1 2 3 4 5 6 7 8 9
в квадрате:         1 4 9 16 25 36 49 64 81
```

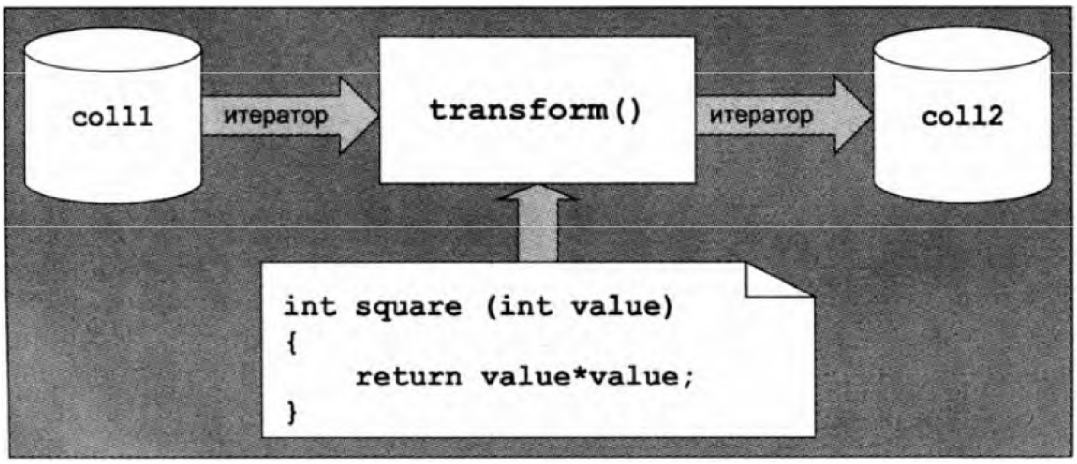


Рис. 6.11. Как работает алгоритм **`transform()`**

6.8.2. Предикаты

Предикат — это особая разновидность вспомогательной функции. Предикаты возвращают булево значение и часто используются для задания критерия сортировки или поиска. В зависимости от поставленной цели предикаты бывают унарными и бинарными.

Не каждая унарная или бинарная функция, возвращающая булево значение, является корректным предикатом. Кроме того, библиотека STL требует, чтобы предикаты не имели состояния, т.е. они всегда должны возвращать один и тот же результат для одного и того же значения. Это исключает применение функций, модифицирующих свое внутреннее состояние при вызове (см. раздел 10.1.4).

Унарные предикаты

Унарные предикаты проверяют конкретное свойство единственного аргумента. Типичным примером является функция, используемая как критерий поиска для обнаружения первого простого числа.

```
// stl/prime1.cpp

#include <list>
#include <algorithm>
#include <iostream>
#include <cstdlib> // для abs()
using namespace std;

// предикат, возвращающий результат проверки, является ли целое число простым
bool isPrime (int number)
{
   // игнорируем отрицательные числа
   number = abs(number);

   // 0 и 1 - не простые числа
   if (number == 0 || number == 1) {
       return false;
   }

   // находим делитель, деление на который происходит без остатка
   int divisor;
   for (divisor = number/2; number%divisor != 0; --divisor) {
       ;
   }

   // если не обнаружены делители больше единицы, значит, число простое
   return divisor == 1;
}

int main()
{
   list<int> coll;

   // вставляем элементы от 24 до 30
   for (int i=24; i<=30; ++i) {
```

```
        coll.push_back(i);
    }

    // ищем простое число
    auto pos = find_if (coll.cbegin(), coll.cend(), // диапазон
                        isPrime);                   // предикат
    if (pos != coll.end()) {
        // найден
        cout << *pos << " - простое число" << endl;
    }
    else {
        // не найден
        cout << "простое число не найдено" << endl;
    }
}
```

В этом примере для поиска первого элемента в заданном диапазоне, для которого передаваемый предикат возвращает `true`, используется алгоритм `find_if()`. Здесь предикатом является функция `isPrime()`, проверяющая, является ли число простым. Используя этот предикат, алгоритм возвращает первое простое число в заданном диапазоне. Если в диапазоне не найден ни один диапазон, соответствующий предикату, алгоритм возвращает конец диапазона (второй аргумент). Это условие проверяется после выполнения вызова. Коллекция в данном примере содержит простое число в диапазоне от 24 до 30, поэтому результат работы программы выглядит следующим образом:

```
29 - простое число
```

Бинарные предикаты

Бинарные предикаты обычно сравнивают конкретное свойство двух аргументов. Например, для сортировки элементов в соответствии с заданным пользователем критерием можно написать простую предикатную функцию. Это может оказаться необходимым в ситуациях, в которых операция < неприемлема или когда нужен другой критерий.

В следующем примере сортируются элементы дека по фамилии и имени человека:

```
// stl/sort1.cpp

#include <algorithm>
#include <deque>
#include <string>
#include <iostream>
using namespace std;

class Person {
    public:
        string firstname() const;
        string lastname() const;
        ...
    };

// бинарный предикат:
// - возвращает результат проверки, предшествует ли один человек другому
```

```
bool personSortCriterion (const Person& p1, const Person& p2)
{
   // человек предшествует другому человеку,
   // - если его фамилия предшествует фамилии другого человека
   // - если фамилии совпадают, а имя первого человека
   //   предшествует имени второго
   return p1.lastname()<p2.lastname() ||
          (p1.lastname()==p2.lastname() &&
          p1.firstname()<p2.firstname());
   }

int main()
{
   deque<Person> coll;
   ...
   sort(coll.begin(),coll.end(), // диапазон
        personSortCriterion);    // критерий сортировки
   ...
}
```

Отметим, что критерий сортировки можно реализовать как функциональный объект. Преимуществом этого вида реализации является то, что критерием является тип, который можно использовать, например, для объявления множеств, использующих этот критерий для сортировки элементов. Реализация такого критерия сортировки описана в разделе 10.1.1.

6.9. Использование лямбда-выражений

Лямбда-выражения, введенные в стандарте C++11, описывают функциональное поведение в виде выражения или оператора (см. раздел 3.1.10). Благодаря этому можно определять объекты, выражающие функциональное поведение, и передавать их в качестве подставляемых аргументов в алгоритмы для использования в качестве предикатов и для других целей.

Например, рассмотрим такой код:

```
// возводим все элементы в куб
std::transform (coll.begin(), coll.end(), // источник
                coll.begin(),             // назначение
                [](double d) {            // лямбда как функтор
                   return d*d*d;
                });
```

Выражение

```
[](double d) { return d*d*d; }
```

определяет лямбда-выражение, представляющее функциональный объект, возвращающий число типа `double`, возведенное в куб. Как видим, это дает возможность задать функциональное поведение, непосредственно передаваемое алгоритму `transform()` при его вызове.

Преимущества лямбда-функций

Использование лямбда-функций для определения поведения в рамках библиотеки STL устраняет много недостатков прошлых вариантов реализации такой функциональности. Допустим, требуется найти первый элемент в коллекции, имеющий значение больше x и меньше y:

```
// stl/lambda1.cpp

#include <algorithm>
#include <deque>
#include <iostream>
using namespace std;

int main()
{
   deque<int> coll = { 1, 3, 19, 5, 13, 7, 11, 2, 17 };
   int x = 5;
   int y = 12;
   auto pos = find_if (coll.cbegin(), coll.cend(), // диапазон
                       [=](int i) {                // критерий поиска
                           return i > x && i < y;
                       });
   cout << "первый элемент >5 и <12: " << *pos << endl;
}
```

Вызывая алгоритм `find_if()`, мы передаем соответствующий предикат в качестве третьего аргумента.

```
auto pos = find_if (coll.cbegin(), coll.cend(),
                    [=](int i) {
                        return i > x && i < y;
                    });
```

Лямбда-функция — это просто функциональный объект, получающий целое число `i` и возвращающий результат проверки, больше ли число `i` числа `x` и меньше ли оно числа `y`.

```
[=](int i) {
    return i > x && i < y;
}
```

Указывая `=` в качестве *захвата* внутри конструкции `[=]`, мы передаем в тело лямбда-функции *по значению* символы, которые были корректны в момент объявления лямбда-функции. Таким образом, в лямбда-функции мы можем читать переменные `x` и `y`, объявленные в функции `main()`. Если используется конструкция `[&]`, то значения можно передавать по ссылке, так что в теле лямбда-функции эти значения можно модифицировать (детали захвата описаны в разделе 3.1.10).

Сравним теперь этот способ поиска первого элемента, удовлетворяющего условию ">5 и <12", с другими подходами, использовавшимися в языке C++ до появления лямбда-функций.

- В отличие от циклов, написанных вручную,

```
// первый элемент, удовлетворяющий условию "> x и < y"
vector<int>::iterator pos;
for (pos = coll.begin() ; pos != coll.end(); ++pos) {
     if (*pos > x && *pos < y) {
         break; // the loop
     }
}
```

  лямбда-функции позволяют использовать стандартные алгоритмы и избегать неуклюжей конструкции `break`.

- В отличие от предикатов, написанных вручную,

```
bool pred (int i)
{
     return i > x && i < y;
}
...
pos = find_if (coll.begin(), coll.end(), // диапазон
               pred);                    // критерий поиска
```

  лямбда-выражения не создают проблем, которые возникают, когда детали поведения определены где-то в другом месте и приходится выяснять, что именно ищет алгоритм `find_if()`, если в программе нет точных комментариев. Кроме того, компиляторы языка C++ лучше оптимизируют лямбда-функции, чем обычные функции.

Еще важнее то, что доступ к переменным `x` и `y` в этом сценарии действительно крайне неудобен. До появления стандарта C++11 обычно в таких ситуациях использовались функциональные объекты (см. раздел 6.10), ярко демонстрирующие неуклюжесть такого подхода:

```
class Pred
{
   private:
      int x;
      int y;
   public:
      Pred (int xx, int yy) : x(xx), y(yy) {
      }
   bool operator() (int i) const {
      return i > x && i < y;
   }
};
...
pos = find_if (coll.begin(), coll.end(), // диапазон
               Pred(x,y));               // критерий поиска
```

- В отличие от применения связывателей (см. раздел 6.10.3),

```
pos = find_if (coll.begin(), coll.end(), // диапазон
               bind(logical_and<bool>(), // критерий поиска
```

```
                bind(greater<int>(),_1,x),
                bind(less<int>(),_1,y)));
```

выражения становятся много понятнее.

Итак, лямбда-функции впервые обеспечивают удобный, понятный, быстрый и технологичный подход к использованию алгоритмов STL.

Использование лямбда-функций в качестве критерия сортировки

В качестве другого примера продемонстрируем использование лямбда-выражений для определения критерия сортировки вектора элементов типа `Person` (соответствующая программа, в которой используется функция, определяющая критерий сортировки, приведена в разделе 6.8.2).

```
// stl/sort2.cpp

#include <algorithm>
#include <deque>
#include <string>
#include <iostream>
using namespace std;
class Person {
   public:
      string firstname() const;
      string lastname() const;
      ...
};

int main()
{
   deque<Person> coll;
   ...

   // сортируем вектор типа Persons по фамилии (или имени):
   sort(coll.begin(),coll.end(),                    // диапазон
        [] (const Person& p1, const Person& p2) { // критерий сортировки
            return p1.lastname()<p2.lastname() ||
                   (p1.lastname()==p2.lastname() &&
                    p1.firstname()<p2.firstname());
        });
   ...
}
```

Ограничения лямбда-функций

Тем не менее лямбда-функции не всегда являются наилучшим средством. Рассмотрим, например, использование лямбда-функции для определения критерия сортировки для ассоциативных массивов:

```
auto cmp = [] (const Person& p1, const Person& p2) {
               return p1.lastname()<p2.lastname() ||
```

```
                    (p1.lastname()==p2.lastname()&&
                     p1.firstname()<p2.firstname());
        };
...
std::set<Person,decltype(cmp)> coll(cmp);
```

Поскольку в объявлении объекта класса `set` необходимо определить тип лямбда-функции, приходится использовать операцию `decltype` (см. раздел 3.1.11), возвращающую тип лямбда-объекта, например `cmp`. Отметим, что необходимо также передать лямбда-объект в конструктор `coll`; в противном случае класс `coll` будет вынужден вызвать конструктор по умолчанию для передаваемого критерия сортировки, в то время как лямбда-функции не могут иметь конструкторов по умолчанию и операций присваивания[11]. Итак, для критерия сортировки класс, определяющий функциональные объекты, может оказаться более интуитивно понятным.

Другая проблема, связанная с лямбда-функциями, заключается в том, что лямбда-функция не может иметь внутреннего состояния, которое могло бы сохраняться при многократных вызовах. Если требуется такое состояние, то необходимо объявить объект или переменную во внешней области видимости и передать ее по ссылке в захват лямбда-функции. По контрасту функциональные объекты позволяют инкапсулировать внутреннее состояние (см. раздел 10.3.2).

Тем не менее лямбда-функции можно использовать для определения хеш-функций и/или критерия эквивалентности для неупорядоченных контейнеров. Пример приведен в разделе 7.9.7.

6.10. Функциональные объекты

Функциональные аргументы алгоритмов не обязаны быть функциями. На примере лямбда-выражений видно, что функциональными аргументами могут быть объекты, ведущие себя как функции. Такие объекты называются *функциональными объектами*, или *функторами*[12]. Вместо использования лямбда-функции можно определить функциональный объект как объект класса, содержащего *операцию вызова функции*. Это было возможно еще до появления стандарта C++11.

6.10.1. Определение функциональных объектов

Функциональные объекты — это еще один пример проявления мощи обобщенного программирования и концепции чистой абстракции. Все, что *ведет* себя как функция, является *функцией*. Итак, если определить объект, ведущий себя как функция, то его можно использовать как функцию.

Каково же поведение функции? Функциональное поведение — это нечто, что можно вызывать с помощью пары скобок и передачи аргументов. Например:

---

[11] Благодарю за это замечание Алисдара Мередита (Alisdair Meredith).

[12] В соответствии со стандартом C++11 *функциональным объектом* считается любой объект, который может быть использован в качестве вызова функции. Таким образом, указатели на функции, объекты классов, содержащие операцию `operator()` или преобразование в указатель на функцию, а также лямбда-функции являются функциональными объектами. Однако в книге этот термин применяется только к объектам класса, содержащего оператор `operator()`.

```
function(arg1,arg2); // вызов функции
```

Если требуется, чтобы объекты вели себя подобным образом, необходимо сделать возможным вызов этих объектов с помощью пары скобок и передачи аргументов. Да, это возможно (в языке C++ возможно почти все). Для этого достаточно объявить операцию () с соответствующими типами параметров.

```
class X {
   public:
      // определение операции "вызов функции":
      return-value operator() (arguments) const;
      ...
};
```

Теперь объекты этого класса можно использовать как вызов функции

```
X fo;
...
fo(arg1,arg2); // вызов operator() из объекта-функции fo
```

Этот вызов эквивалентен конструкции.

```
fo.operator()(arg1,arg2); // вызов operator() из функционального объекта fo
```

Рассмотрим законченный пример. Это вариант функционального объекта из предыдущего примера (см. раздел 6.8.1), который делал то же самое с помощью обычной функции:

```
// stl/foreach2.cpp

#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

// простой функциональный объект, который выводит на экран
// передаваемые аргументы
class PrintInt {
   public:
      void operator() (int elem) const {
         cout << elem << ' ';
      }
};

int main()
{
   vector<int> coll;

   // вставляем элементы от 1 до 9
   for (int i=1; i<=9; ++i) {
        coll.push_back(i);
   }

   // выводим на экран все элементы
   for_each (coll.cbegin(), coll.cend(), // диапазон
```

```
            PrintInt());                    // операция
   cout << endl;
}
```

Класс `PrintInt` определяет объекты, к которым можно применять операцию `()` с аргументом типа `int`. Выражение `PrintInt()` в операторе

```
for_each (coll.cbegin(), coll.cend(), PrintInt());
```

создает временный объект этого класса, который передается алгоритму `for_each()` в качестве аргумента.

Алгоритм `for_each()` может быть записан следующим образом:

```
namespace std {
   template <typename Iterator, typename Operation>
   Operation for_each (Iterator act, Iterator end, Operation op)
   {
      while (act != end) { // пока не достигнут конец
         op(*act); // - вызываем op() с текущим элементом
         ++act;    // - перемещаем итератор на следующий элемент
      }
      return op;
   }
}
```

Алгоритм `for_each()` использует временный функциональный объект `op` при вызове `op(*act)` для каждого элемента `act`. Если бы `op` была обычной функцией, то алгоритм `for_each()` просто вызвал бы ее с аргументом `*act`. Если же `op` — функциональный объект, то алгоритм `for_each()` вызывает ее операцию `()` с аргументом `*act`. Таким образом, в этом примере алгоритм `for_each()` выполняет следующий вызов:

```
PrintInt::operator()(*act)
```

Может возникнуть вопрос: зачем все это нужно? Функциональные объекты могут даже показаться странными, уродливыми и бессмысленными. Действительно, они усложняют код. Однако функциональные объекты — это нечто большее, чем просто функции, и благодаря этому они имеют определенные преимущества.

1. **Функциональный объект — это "функция с состоянием"**. Объекты, ведущие себя как указатели, называются интеллектуальными указателями. Это же относится к объектам, ведущим себя как функции: они могут рассматриваться как "интеллектуальные функции", потому что обладают дополнительными возможностями помимо операции `()`. Функциональные объекты могут содержать другие функции-члены и атрибуты. Это означает, что функциональные объекты имеют состояние. Фактически одна и та же функциональная возможность, выраженная двумя разными функциональными объектами одного и того же типа, в одно и то же время может иметь разные состояния. Для обычных функций это невозможно. Другим преимуществом функциональных объектов является возможность их инициализации в ходе выполнения программы до вызова.
2. **Каждый функциональный объект имеет свой тип**. Обычные функции имеют разные типы, только если их сигнатуры отличаются друг от друга. Однако функциональные объекты могут иметь разные типы, даже если их сигнатуры совпадают.

Фактически каждое функциональное поведение, определенное с помощью функционального объекта, имеет свой собственный тип. Это важное улучшение для обобщенного программирования с помощью шаблонов, потому что теперь появляется возможность передавать функциональное поведение в качестве шаблонного параметра. В результате контейнеры разных типов могут использовать один и тот же вид функциональных объектов в качестве критерия сортировки, предотвращая присваивание, объединение и сравнение коллекций, имеющих разные критерии сортировки. Можно даже разрабатывать иерархии функциональных объектов, чтобы, например, создавать конкретные варианты общего критерия.

3. **Функциональные объекты, как правило, работают быстрее, чем обычные функции**. Концепция шаблонов обычно позволяет улучшить оптимизацию, поскольку больше деталей определяется на этапе компиляции. Таким образом, передача функциональных объектов вместо обычных функций часто повышает производительность программы.

В оставшейся части раздела рассматриваются примеры, демонстрирующие преимущество функциональных объектов над обычными функциями. В главе 10, посвященной исключительно функциональным объектам, приведено больше примеров и подробностей. В частности, показано, как извлечь выгоду из возможности передавать функциональное поведение в качестве шаблонного параметра.

Допустим, что мы хотим добавить определенное значение ко всем элементам коллекции. Если это значение известно на этапе компиляции, то можно использовать обычную функцию.

```
void add10 (int& elem)
{
   elem += 10;
}

void f1()
{
   vector<int> coll;
   ...

   for_each (coll.begin(), coll.end(), // диапазон
             add10);                   // операция
}
```

Если необходимо добавлять разные значения, известные на этапе компиляции, то можно использовать шаблон.

```
template <int theValue>
void add (int& elem)
{
   elem += theValue;
}

void f1()
{
   vector<int> coll;
   ...
```

```
   for_each (coll.begin(), coll.end(), // диапазон
             add<10>);                 // операция
}
```

Если значение добавляется на этапе выполнения программы, ситуация осложняется. В этом случае необходимо передавать значение в функцию до того, как она будет вызвана. Как правило, для этого используется глобальная переменная, которая используется как функцией, вызывающей алгоритм, так и функцией, которая вызывается алгоритмом, добавляющим значение. Это плохой стиль программирования.

Если такая функция потребуется дважды, чтобы добавить два разных значения, и оба этих значения должны добавляться на этапе выполнения программы, одной обычной функцией уже не обойтись. Придется либо передать дескриптор, либо написать две разные функции. Вы копировали когда-нибудь функцию, содержащую статическую переменную для хранения ее состояния, просто потому, что вам потребовалась точно такая же функция с другим состоянием в тот же момент времени? Эта проблема относится к той же категории.

Работая с функциональными объектами, можно написать более интеллектуальную функцию, ведущую себя желательным образом. Поскольку она может иметь состояние, объект может быть инициализирован правильным значением. Рассмотрим законченный пример[13]:

```
// stl/add1.cpp

#include <list>
#include <algorithm>
#include <iostream>
#include "print.hpp"
using namespace std;

// функциональный объект, добавляющий значение, полученное при инициализации
class AddValue {
   private:
      int theValue; // добавляемое значение
   public:
      // конструктор инициализирует добавляемое значение
      AddValue(int v) : theValue(v) {
   }

   // "вызов функции" для элемента, к которому добавляется значение
   void operator() (int& elem) const {
      elem += theValue;
   }
};

int main()
{
   list<int> coll;

   // вставляем элементы от 1 до 9
```

[13] Вспомогательная функция PRINT_ELEMENTS() была введена в разделе 6.6.

```
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    PRINT_ELEMENTS(coll,"инициализирован: ");

    // добавляем значение 10 к каждому элементу
    for_each (coll.begin(), coll.end(), // диапазон
              AddValue(10));            // операция

    PRINT_ELEMENTS(coll,"после добавления 10: ");

    // добавляем значение первого элемента к каждому элементу
    for_each (coll.begin(), coll.end(), // диапазон
              AddValue(*coll.begin())); // операция

    PRINT_ELEMENTS(coll,"после добавления первого элемента: ");
}
```

После инициализации коллекция содержит значения от 1 до 9:

```
инициализирован: 1 2 3 4 5 6 7 8 9
```

Первый вызов алгоритма `for_each()` добавляет 10 к каждому значению:

```
for_each (coll.begin(), coll.end(), // диапазон
          AddValue(10));            // операция
```

Здесь выражение `AddValue(10)` создает объект типа `AddValue`, инициализированный значением 10. Конструктор `AddValue` сохраняет это значение как член `theValue`. В алгоритме `for_each()` вызывается операция `()` для каждого элемента коллекции `coll`. Этот вызов операции `()` относится к передаваемому временному функциональному объекту типа `AddValue`. Текущий элемент передается как аргумент.

Функциональный объект добавляет свое значение 10 к каждому элементу. После этого элементы принимают следующие значения:

```
после добавления 10: 11 12 13 14 15 16 17 18 19
```

Второй вызов алгоритма `for_each()` использует ту же самую функциональную возможность для добавления значения первого элемента ко всем элементам коллекции. Этот вызов инициализирует временный функциональный объект типа `AddValue` первым элементом коллекции.

```
AddValue(*coll.begin())
```

Результат работы программы принимает следующий вид:

```
после добавления первого элемента: 22 23 24 25 26 27 28 29 30
```

Усовершенствованный вариант этого примера, в котором тип функционального объекта `AddValue` является шаблонным типом добавляемого значения, рассматривается в разделе 11.4.

Используя этот прием, можно решить проблему с помощью двух разных функциональных объектов, имеющих разные состояния в один и тот же момент времени. Например, можно просто объявить функциональные объекты и использовать их независимо друг от друга.

```
AddValue addx(x); // функциональный объект, добавляющий значение x
AddValue addy(y); // функциональный объект, добавляющий значение y
for_each (coll.begin(),coll.end(), // добавляем значение x к каждому элементу
          addx);
...
for_each (coll.begin(),coll.end(), // добавляем значение y к каждому элементу
          addy);
...
for_each (coll.begin(),coll.end(), // добавляем значение x к каждому элементу
          addx);
```

Аналогично можно предусмотреть дополнительную функцию-член для запроса или изменения состояния функционального объекта на протяжении срока его существования. Хороший пример приведен в разделе 10.1.3.

Отметим, что для некоторых алгоритмов стандартная библиотека языка не регламентирует, как часто функциональные объекты могут вызываться для каждого элемента, и может так случиться, что элементам будут передаваться разные копии функционального объекта. Если функциональные объекты используются как предикаты, это может вызвать ужасные последствия. Эта тема освещается в разделе 10.1.4.

6.10.2. Стандартные функциональные объекты

Стандартная библиотека языка C++ содержит несколько стандартных функциональных объектов, выполняющих основные операции. Благодаря им в определенных ситуациях можно обойтись без создания собственных функциональных объектов. Типичным примером является функциональный объект, используемый как критерий сортировки. Критерий сортировки с помощью операции < представляет собой стандартный функциональный объект `less<>`. Таким образом, объявление

```
set<int> coll;
```

разворачивается в объявление

```
set<int,less<int>> coll; // сортируем элементы с помощью операции <
```

Теперь легко упорядочить элементы в обратном порядке.

```
set<int,greater<int>> coll; // сортируем элементы с помощью операции >
```

Другим примером применения стандартных функциональных объектов являются алгоритмы. Рассмотрим следующую программу:

```
// stl/fo1.cpp

#include <deque>
#include <algorithm>
#include <functional>
```

```
#include <iostream>
#include "print.hpp"
using namespace std;

int main()
{
   deque<int> coll = { 1, 2, 3, 5, 7, 11, 13, 17, 19 };

   PRINT_ELEMENTS(coll,"инициализация: ");

   // меняем знаки всех значений в коллекции coll на противоположный
   transform (coll.cbegin(),coll.cend(), // источник
              coll.begin(),              // назначение
              negate<int>());            // операция
   PRINT_ELEMENTS(coll,"смена знака: ");

   // возводим все значения в коллекции coll в квадрат
   transform (coll.cbegin(),coll.cend(), // первый источник
              coll.cbegin(),             // второй источник
              coll.begin(),              // назначение
              multiplies<int>());        // операция
   PRINT_ELEMENTS(coll,"в квадрате: ");
}
```

Сначала в программу включается заголовок для стандартных объектов-функций: `<functional>`

```
#include <functional>
```

Затем два стандартных функциональных объекта используются для замены знака и возведения в квадрат элементов коллекции `coll`. Во фрагменте

```
transform (coll.cbegin(), coll.cend(), // источник
           coll.begin(),               // назначение
           negate<int>());             // операция
```

выражение

```
negate<int>()
```

создает объект-функцию стандартного шаблонного класса `negate<>`, который просто возвращает элемент типа `int`, имеющий противоположный знак по отношению к тому элементу, для которого он был вызван. Алгоритм `transform()` использует эту операцию для преобразования всех элементов первой коллекции во вторую. Если диапазоны источника и назначения совпадают, как в данном случае, возвращаемые элементы с противоположным знаком заменяют исходные элементы. Таким образом, данный оператор изменяет на противоположные знаки всех элементов коллекции.

Аналогично функциональный объект `multiplies` используется для возведения в квадрат всех элементов коллекции `coll`.

```
transform (coll.cbegin(), coll.cend(), // первый источник
           coll.cbegin(),              // второй источник
```

```
        coll.begin(),                // назначение
        multiplies<int>());          // операция
```

Здесь другая форма алгоритма `transform()` объединяет элементы двух коллекций с помощью заданной операции и записывает результат в третью коллекцию. Как и раньше, все коллекции совпадают, поэтому каждый элемент умножается на себя, а результаты заменяют старые значения.

Итак, программа выводит на экран следующие строки:

```
инициализация: 1 2 3 5 7 11 13 17 19
смена знака:  -1 -2 -3 -5 -7 -11 -13 -17 -19
в квадрате:    1 4 9 25 49 121 169 289 361
```

6.10.3. Связыватели

Для объединения функциональных объектов с другими значениями и выполнения других специфических задач используются *функциональные адаптеры*, или *связыватели* (binders). Рассмотрим законченный пример.

```
// stl/bind1.cpp

#include <set>
#include <deque>
#include <algorithm>
#include <iterator>
#include <functional>
#include <iostream>
#include "print.hpp"
using namespace std;
using namespace std::placeholders;

int main()
{
   set<int,greater<int>> coll1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
   deque<int> coll2;

   // Примечание: благодаря критерию сортировки greater<>()
   // элементы следуют в обратном порядке:
   PRINT_ELEMENTS(coll1,"инициализация: ");

   // преобразовываем все элементы в коллекцию coll2,
   // умножая их на 10
   transform (coll1.cbegin(),coll1.cend(),     // источник
              back_inserter(coll2),            // назначение
              bind(multiplies<int>(),_1,10)); // операция
   PRINT_ELEMENTS(coll2,"преобразование: ");

   // заменяем значение, равное 70, значением, равным 42
   replace_if (coll2.begin(),coll2.end(),      // диапазон
               bind(equal_to<int>(),_1,70),    // диапазон замены
               42);                            // новое значение
   PRINT_ELEMENTS(coll2,"замена: ");
```

```
    // удаляем все элементы со значениями от 50 до 80
    coll2.erase(remove_if(coll2.begin(),coll2.end(),
                bind(logical_and<bool>(),
                bind(greater_equal<int>(),_1,50),
                bind(less_equal<int>(),_1,80))),
                coll2.end());
    PRINT_ELEMENTS(coll2,"удаление: ");
}
```

Здесь оператор

```
transform (coll1.cbegin(),coll1.cend(),     // источник
           back_inserter(coll2),            // назначение
           bind(multiplies<int>(),_1,10)); // операция
```

преобразовывает все элементы коллекции `coll1` в коллекцию `coll2` (вставляя их), умножая каждый элемент на 10. Для определения соответствующей операции используется функциональный адаптер `bind()`, позволяющий создавать высокоуровневые функциональные объекты из низкоуровневых функциональных объектов и шаблонных заготовок, представляющих собой числовые идентификаторы, начинающиеся с символа подчеркивания. Оператор

```
bind(multiplies<int>(),_1,10)
```

определяет функциональный объект, умножающий первый переданный аргумент на 10.

Аналогичный функциональный объект можно было бы использовать для умножения на 10 всех элементов. Например, следующие инструкции записывают в стандартный поток вывода число 990:

```
auto f = bind(multiplies<int>(),_1,10);
cout << f(99) << endl;
```

Этот функциональный объект передается алгоритму `transform()`, который ожидает в качестве четвертого аргумента операцию, получающую один аргумент, а именно текущий элемент. Впоследствии алгоритм `transform()` выполняет операцию "умножить на 10" для каждого элемента и вставляет результат в коллекцию `coll2`, т.е. после всех элементов коллекции `coll2` следуют все значения коллекции `coll1`, умноженные на 10.

Аналогично в вызове

```
replace_if (coll2.begin(),coll2.end(),     // диапазон
            bind(equal_to<int>(),_1,70),   // критерий замены
            42);                           // новое значение
```

в качестве критерия для выбора элемента, который должен быть заменен числом 42, используется следующий функциональный адаптер.

```
bind(equal_to<int>(),_1,70)
```

Здесь функциональный адаптер `bind()` вызывает бинарный предикат `equal_to`, которому в качестве первого параметра передается первый аргумент, а в качестве второго параметра — число 70. Таким образом, функциональный объект, определенный с помощью связывателя `bind()`, возвращает `true`, если переданный аргумент (элемент коллекции `coll2`) равен 70. В результате оператор заменяет все значения, равные 70, значением 42.

Последний пример использует комбинацию связывателей, где оператор

```
bind(logical_and<bool>(),
     bind(greater_equal<int>(),_1,50),
     bind(less_equal<int>(),_1,80))
```

задает для параметра x унарный предикат "x>=50&&x<=80." Этот пример демонстрирует возможность использования вложенных связывателей bind() для описания более сложных предикатов и функциональных объектов. В данном случае алгоритм remove_if() использует функциональный объект для удаления из коллекции всех значений, лежащих в диапазоне от 50 до 80. Фактически алгоритм remove_if() лишь изменяет порядок и возвращает новый конец диапазона, а функция-член coll2.erase() удаляет "удаленные" элементы из коллекции coll2 (см. детали в разделе 6.7.1).

Результат работы программы выглядит следующим образом:

```
инициализация:  9 8 7 6 5 4 3 2 1
преобразование: 90 80 70 60 50 40 30 20 10
замена:         90 80 42 60 50 40 30 20 10
удаление:       90 42 40 30 20 10
```

Отметим, что заполнители имеют свое пространство имен: std::placeholders. По этой причине в начало программы вставлена соответствующая директива, позволяющая задавать в качестве первого или второго параметра связывателя заполнители _1 или _2. Без использования этой директивы связыватели пришлось бы определять следующим образом:

```
std::bind(std::logical_and<bool>(),
          std::bind(std::greater_equal<int>(),std::placeholders::_1,50),
          std::bind(std::less_equal<int>(),std::placeholders::_1,80))
```

Этот вид программирования называется *функциональной композицией* (functional composition). Интересно, что все эти функциональные объекты обычно объявляются встраиваемыми. Таким образом, мы используем функциональную абстракцию и при этом получаем высокую производительность.

Существуют и другие способы определения функциональных объектов. Например, для вызова функции-члена для каждого элемента коллекции можно использовать вызов

```
for_each (coll.cbegin(), coll.cend(), // диапазон
          bind(&Person::save,_1));    // операция: Person::save(elem)
```

Функциональный объект bind связывает указанную функцию-член, чтобы она вызывалась для каждого элемента, который передается с помощью заполнителя _1. Таким образом, для каждого элемента коллекции coll вызывается функция-член save() из класса Person. Разумеется, все это работает только при условии, что элементы имеют тип Person или тип, производный от типа Person.

Более подробно все стандартные функциональные объекты, функциональные адаптеры и аспекты функциональной композиции рассматриваются в разделе 10.2. Кроме того, там же объясняется, как написать свои собственные функциональные объекты.

До появления документа TR1 для функциональной композиции использовались другие связыватели и адаптеры: bind1st(), bind2nd(), ptr_fun(), mem_fun() и mem_fun_ref(), которые в стандарте C++11 объявлены устаревшими. Детали описаны в разделе 10.2.4.

6.10.4. Функциональные объекты и связыватели против лямбда-функции

Лямбда-функции — это разновидность неявно определенного функционального объекта. Как указано в разделе 6.9, лямбда-функции обычно обеспечивают более интуитивный подход к определению функционального поведения алгоритмов из библиотеки STL. Кроме того, лямбда-функции работают быстрее функциональных объектов.

Тем не менее у лямбда-функций есть несколько недостатков.

- У такого функционального объекта нет скрытого внутреннего состояния. Вместо этого все данные, определяющие его состояние, определяются вызывающей функцией и передаются в виде захвата.
- Исчезает преимущество частичного описания функционального поведения, которое требуется в разных местах. Вы можете определить лямбда-функцию, а затем присвоить ее объекту `auto` (см. раздел 6.9), но читабельность такого определения довольно сомнительна.

6.11. Элементы контейнеров

Элементы контейнеров должны удовлетворять определенным требованиям, поскольку контейнеры обрабатывают их определенным образом. В этом разделе мы опишем эти требования и обсудим последствия того факта, что контейнеры неявно создают копии своих элементов.

6.11.1. Требования к элементам контейнеров

Контейнеры, итераторы и алгоритмы библиотеки STL являются шаблонами. Вследствие этого они могут работать как со стандартными, так и с пользовательскими типами. Однако операции, которые они вызывают, накладывают на них определенные ограничения. Элементы контейнеров STL должны удовлетворять трем основным требованиям.

1. Элемент должен допускать *копирование* или *перемещение*. Таким образом, тип элемента явно или неявно должен иметь копирующий или перемещающий конструктор.
2. Сгенерированная копия должна быть эквивалентной оригиналу. Это значит, что любая проверка равенства должна возвращать одинаковые результаты как для оригинала, так и для копии.
3. Элемент должен допускать *(перемещающее) присваивание*. Контейнеры и алгоритмы используют операции присваивания для перезаписи старых элементов новыми.
4. Элемент должен допускать *удаление* с помощью деструктора. Контейнеры должны удалять внутренние копии элементов, когда эти элементы удаляются из контейнера. Таким образом, деструктор не должен быть закрытым. Кроме того, как обычно в языке C++, деструктор не должен генерировать исключение; в противном случае возникает неопределенное поведение.

Эти три операции неявно генерируются для любого класса. Таким образом, класс автоматически удовлетворяет перечисленным требованиям, если в нем не определены

специальные версии этих операций и нет специальных функций-членов, нарушающих правильную работу этих операций.

Элементы контейнеров должны также удовлетворять следующим требованиям.

- Для некоторых функций-членов последовательных контейнеров должен быть доступным *конструктор, заданный по умолчанию*. Например, можно создать непустой контейнер или увеличить количество элементов в контейнере, не имея представления о том, какое значения должны принимать новые значения. Эти элементы создаются без аргументов с помощью вызова конструктора, заданного по умолчанию для данного типа.
- Для некоторых операций необходимо определять *проверку равенства* с помощью операции ==. Это особенно необходимо при поиске элементов. Однако для неупорядоченных контейнеров можно предусмотреть собственное определение эквивалентности, если их элементы не поддерживают операцию == (см. раздел 7.9.7).
- Элементы ассоциативных контейнеров должны поддерживать операции *критерии сортировки*. По умолчанию в качестве таковой используется операция <, которая вызывается функциональным объектом `less<>`.
- Элементы неупорядоченных контейнеров должны предусматривать *хеш-функцию* и *критерий эквивалентности*. Детали описаны в разделе 7.9.2.

6.11.2. Семантика значений и семантика ссылок

Как правило, все контейнеры создают внутренние копии своих элементов и возвращают их, а не оригиналы. Это означает, что элементы контейнера равны, но не идентичны объектам, помещенным в контейнер. Если модифицировать объект как элемент контейнера, то изменения коснутся лишь копии, а не оригинала.

Копирование объектов означает, что контейнеры STL реализуют *семантику значений*. Контейнеры содержат значения вставленных объектов, а не сами объекты. Однако на практике нам часто нужна *семантика ссылок*. Это значит, что контейнеры содержат ссылки на объекты и именно ссылки являются их элементами.

Ориентация библиотеки STL исключительно на семантику значений имеет как преимущества, так и недостатки. Перечислим преимущества.

- Копирование элементов выполняется просто.
- Ссылки провоцируют ошибки. Постоянно необходимо следить за тем, чтобы ссылки, ссылающиеся на несуществующий объект, уничтожались. Требуется также уметь справляться с циклическими ссылками, которые могут возникнуть.

Недостатки описаны ниже.

- Копирование элементов может снизить производительность работы программы или вообще быть невозможным.
- Управление одним и тем же объектом в разных контейнерах в одно и то же время невозможно.

На практике необходимы оба подхода; нужны и копии, не зависящие от оригиналов (семантика значений), и копии, ссылающиеся на оригинал и соответственно изменяющиеся (семантика ссылок). К сожалению, стандартная библиотека C++ не поддерживает

семантику ссылок. Однако семантику ссылок можно реализовать самостоятельно в терминах семантики значений.

Очевидный подход к реализации семантики ссылок — использование в качестве элементов контейнеров указателей[14]. Однако обычные указатели порождают обычные проблемы. Например, объект, на который он указывает, может больше не существовать, а сравнения могут оказаться некорректными из-за того, что сравниваются указатели, а не объекты. Таким образом, при использовании указателей в качестве элементов контейнера следует быть очень осторожным. Лучше использовать *интеллектуальные указатели*: объекты, имеющие интерфейс указателей, и дополнительные внутренние операции проверки или обработки данных. С появлением документа TR1 стандартная библиотека C++ содержит класс `shared_ptr` для интеллектуальных указателей, которые могут ссылаться на один и тот же объект (см. раздел 5.2.1). Кроме того, можно использовать класс `std::reference_wrapper<>` (см. раздел 5.4.3), позволяющий контейнерам STL содержать ссылки. Примеры обоих подходов описаны в разделе 7.11.

6.12. Ошибки и исключения в библиотеке STL

Ошибки случаются. Они могут быть логическими ошибками, совершенными программистом, или ошибками времени выполнения, вызванными контекстом или средой (например, из-за нехватки памяти). Оба вида этих ошибок можно обработать с помощью исключений. В этом разделе обсуждаются ошибки и исключения, обрабатываемые в библиотеке STL.

6.12.1. Обработка ошибок

Цель разработки библиотеки STL — обеспечить высокую производительность, а не наивысшую безопасность. Обработка ошибок требует времени, поэтому в библиотеке STL она почти не предусмотрена. Это прекрасно, если в программе нет никаких ошибок, и чревато катастрофой, если это не так. Еще когда библиотека STL не была включена в состав стандартной библиотеки C++, велись дискуссии о том, следует ли включать в нее обработку ошибок. Большинство программистов решили, что обработку ошибок включать в библиотеку не следует, ссылаясь на две причины.

1. Проверка ошибок уменьшает производительность, а скорость — главная цель программ. Как уже указывалось, высокая производительность — одна из основных целей разработки библиотеки STL.
2. Если безопасность важнее, чем быстродействие, ее можно обеспечить, либо добавив оболочки, либо используя специальные версии библиотеки STL. Однако, встраивая проверку ошибок во все основные операции, вы снижаете быстродействие программ. Например, если каждая операция по вычислению индекса будет проверять, не выходит ли он за пределы допустимого диапазона, то не сможете написать обращение к элементам массива без проверки этой ошибки. Однако возможно наличие обходного пути.

[14] Программистам на языке C знаком способ реализации семантики ссылок с помощью указателей. В языке C аргументы функции можно передавать только по значению, поэтому для вызова по ссылке используются указатели.

В итоге проверку ошибок в библиотеке STL признали возможной, но не обязательной.

В основу стандартной библиотеки C++ положен следующий принцип: любое нарушение библиотекой STL заданных предусловий приводит к неопределенному поведению. Таким образом, если индексы, итераторы или диапазоны оказываются некорректными, то результат считается неопределенным. Если вы не используете безопасную версию библиотеки STL, то в результате нередко возникает попытка доступа к неопределенной памяти, порождающая ужасные последствия и даже крах программы. В этом смысле библиотека уязвима для ошибок, как указатели в языке C. Поиск таких ошибок может оказаться очень сложным, особенно если не используется безопасная версия библиотеки STL.

В частности, использование библиотеки STL требует выполнения следующих условий.

- Итераторы должны быть корректными. Например, перед использованием они должны быть инициализированы. Отметим, что итераторы могут стать некорректными в результате побочного эффекта от других операций. В частности, итераторы становятся некорректными
  - при вставке, удалении или перемещении элементов векторов и деков;
  - при повторном хешировании неупорядоченных контейнеров (которое также может быть результатом вставки).
- Итераторы, ссылающиеся на запредельную позицию, не ссылаются ни на один конкретный элемент. Таким образом, выполнение операций `*` или `->` для них запрещено. Особенно это относится к значениям, возвращаемым функциями-членами контейнеров `end()`, `cend()` и `rend()`.
- Диапазоны должны быть корректными.
  - Оба итератора, определяющие диапазон, должны ссылаться на один и тот же контейнер.
  - Второй итератор должен быть достижимым из первого.
- Если используется несколько источников, то второй и последующие должны иметь не меньше элементов, чем первый.
- Диапазоны назначения должны иметь достаточное количество элементов, которые можно заменить; в противном случае необходимо использовать итераторы вставки.

Следующий пример демонстрирует несколько возможных ошибок:

```
// stl/iterbug.cpp

#include <vector>
#include <algorithm>
using namespace std;

int main()
{
   vector<int> coll1; // пустая коллекция
   vector<int> coll2; // пустая коллекция

   // ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ПРОГРАММЫ:
   // - начало расположено за концом диапазона
   vector<int>::iterator pos = coll1.begin();
   reverse (++pos, coll1.end());
```

```
    // вставляем элементы от 1 до 9 в коллекцию coll1
    for (int i=1; i<=9; ++i) {
        coll1.push_back (i);
    }

    // ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ПРОГРАММЫ:
    // - замена несуществующих элементов
    copy (coll1.cbegin(), coll1.cend(), // источник
          coll2.begin());               // назначение

    // ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ПРОГРАММЫ:
    // - ошибочные коллекции
    // - cbegin() и cend() ссылаются на разные коллекции
    copy (coll1.cbegin(), coll2.cend(), // источник
          coll1.end());                 // назначение
}
```

Отметим, что, поскольку эти ошибки возникают во время выполнения программы, а не на этапе компиляции, последствия остаются непредсказуемыми.

Существует много способов сделать ошибки при работе с библиотекой STL, причем она никак не пытается предотвратить эти ошибки. Таким образом, целесообразно использовать безопасную библиотеку STL, по крайней мере на этапе разработки программного обеспечения. Первая версия безопасной библиотеки STL была разработана Кеем Хорстманом (Cay Horstmann) [SafeSTL]. Другим примером является библиотека "STLport", доступная без ограничений практически на любой платформе [STLport]. Кроме того, поставщики библиотек стали предусматривать флаги для включения безопасного режима, особенно полезного на этапе разработки программ[15].

6.12.2. Обработка исключений

Библиотека почти никогда не проверяет логические ошибки. Вследствие этого логические проблемы почти никогда не заставляют библиотеку STL генерировать исключения. Фактически только две функции в соответствии со стандартом могут генерировать исключение непосредственно: функция-член `at()`, реализующая версию операции индексирования с проверкой, и функция-член `reserve()`, генерирующая исключение, если переданное количество элементов превышает значение, возвращаемое функцией-членом `max_size()`. В остальных ситуациях стандарт требует лишь, чтобы генерировались обычные стандартные исключения, такие как `bad_alloc`, из-за нехватки памяти или исключения, предусмотренные для пользовательских операций.

Когда генерируются исключения и что при этом происходит с компонентами библиотеки STL? В течение продолжительного процесса выработки стандарта C++98 эти вопросы оставлялись без ответа. Фактически каждое исключение приводило к неопределенному поведению. Даже уничтожение контейнера STL приводило к таковому, если исключение генерировалось во время выполнения одной из операций. Таким образом, библиотека STL была бесполезной, если программисту требовались гарантии и определенное поведение, поскольку не было возможным даже раскручивание стека.

[15] Например, компилятор `gcc` для этой цели предусматривает ключ `-D_GLIBCXX_DEBUG`.

Способ обработки исключений был одним из последних тем, рассмотренных в процессе выработки стандарта C++98. Найти хорошее решение было непросто, и его поиск занял много времени по следующим причинам.

1. Было очень трудно определить степень безопасности, которую должна обеспечить стандартная библиотека C++. Может показаться, что в любом случае следует стремиться к наивысшей степени безопасности. Например, можно предположить, что вставка нового элемента в любую позицию вектора должна быть либо успешной, либо нет. Обычно исключение может возникнуть при копировании элементов вправо, чтобы создать место для нового элемента, и при этом полное восстановление невозможно. Для того чтобы достичь поставленной цели, операция вставки должна выполнить копирование *каждого* элемента вектора в новое хранилище, что значительно снизит производительность программы. Если целью проекта является производительность, как в случае библиотеки STL, идеальную обработку ошибок реализовать не удастся ни при каких условиях. Необходимо искать компромисс между безопасностью и быстродействием.
2. Нет никаких сомнений, что лучше иметь гарантированное, точно определенное поведение исключений без снижения быстродействия, чем рисковать крахом системы. Однако обработка исключений снижает производительность программы. Это может противоречить основной цели проекта — обеспечить максимальное быстродействие. В процессе выработки стандарта C++98 разработчики компилятора утверждали, что в принципе обработку исключений можно реализовать без значительного снижения производительности. Однако оказалось, что спецификации исключений все же снижают производительность, поэтому в стандарте C++11 они были заменены ключевым словом `noexcept` (см. раздел 3.1.7).

В результате всех этих обсуждений стандартная библиотека C++, соответствующая стандарту, дает следующую основную гарантию безопасности исключений: при генерировании исключений стандартная библиотека C++ не порождает утечек ресурсов и не нарушает инварианты контейнеров[16].

К сожалению, во многих ситуациях этого недостаточно. Часто необходимо иметь более сильные гарантии, чем то, что при генерировании исключения операция не будет иметь последствий. С точки зрения исключений такие операции можно рассматривать как *атомарные*. Разумеется, применяя терминологию баз данных, мы могли бы сказать, что эти операции поддерживают принцип *фиксации или отката* (commit-or-rollback), или *безопасность транзакций* (transaction safe).

С учетом этой более сильной гарантии стандартная библиотека C++ обеспечивает следующее.

- Функции `erase()`, `clear()`, `pop_back()`, `pop_front()` или `swap()` в общем случае не генерируют исключений. Кроме того, ни копирующие конструкторы, ни операции присваивания возвращаемого итератора не генерируют исключений.
- Для всех *контейнеров на основе узлов* (списков, односвязных списков, множеств, мультимножеств, отображений и мультиотображений), включая *неупорядоченные контейнеры*, при любом сбое во время создания узла контейнер остается в текущем

[16] Я выражаю горячую благодарность Дэйву Абрамсу (Dave Abrahams) и Грэгу Колвину (Greg Colvin) за их работу по обеспечению безопасности исключений в стандартной библиотеке C++ и сделанные ими ценные замечания.

состоянии. Более того, удаление узла не может приводить к сбою, при условии, что деструкторы не генерируют исключений. Однако при выполнении операции вставки нескольких элементов в ассоциативные контейнеры необходимость сохранения порядка следования элементов делает полное восстановление после генерации исключения непрактичным. Таким образом, все операции вставки единственного элемента в ассоциативные или неупорядоченные контейнеры подчиняются принципу фиксации и отката, при условии, что хеш-функции неупорядоченных контейнеров не генерируют исключений. Иначе говоря, вставка единственного элемента либо оказывается успешной, либо не имеет последствий. Кроме того, гарантируется, что все операции удаления как единственного элемента, так и нескольких элементов одной операцией всегда являются успешными, при условии, что функции сравнения и хеш-функция контейнера не генерируют исключений.

- Для списков даже операции вставки нескольких элементов обеспечивают безопасность транзакций. Фактически все операции над списком, за исключением `remove()`, `remove_if()`, `merge()`, `sort()` и `unique()`, либо оказываются успешными, либо не имеют последствий. Для некоторых исключительных ситуаций стандартная библиотека C++ предусматривает условные гарантии. Таким образом, если вам нужен контейнер, обеспечивающий безопасность транзакций, следует выбирать список.
- Для односвязных списков функции-члены `insert_after()`, `emplace_after()` и `push_front()` обеспечивают безопасность транзакций[17].
- Все *контейнеры на основе массивов* (объекты класса `array`, векторы и деки) не могут полностью восстановить свое состояние после неудачной операции вставки. Для этого пришлось бы перед выполнением любой операции вставки копировать все последующие элементы, что заняло бы много времени. В то же время операции `push` и `pop`, работающие с последним элементом, не требуют копирования существующих элементов. Если они генерируют исключение, то гарантируется, что последствий не будет. Однако, если перемещающий конструктор не гарантирует отсутствие исключений, то для обеспечения такого поведения следует использовать копирующий, а не перемещающий конструктор. Таким образом, спецификации `nothrow` и `noexcept` для операций, перемещающих элементы, повышают быстродействие программы. Более того, если элементы имеют тип, содержащий операции копирования (копирующий конструктор и операцию присваивания), не генерирующие исключений, то любая операция контейнера для этих элементов либо будет успешной, либо не будет иметь последствий.

Отметим, что все эти гарантии основаны на требовании, чтобы деструкторы никогда не генерировали исключения. Стандартная библиотека C++ строго выполняет это требование, и точно так же должны поступать программисты.

Если необходим контейнер, полностью реализующий принцип фиксации и отката, следует выбирать либо список (не вызывая функции `remove()`, `remove_if()`, `merge()`, `sort()` и `unique()`), либо ассоциативный или неупорядоченный контейнер (не вызывая операции одновременной вставки нескольких элементов). Это позволяет избежать создания копий перед выполнением модифицирующей операции, чтобы предотвратить потерю данных. Отметим, что создание копий контейнера может быть очень затратным.

[17] Стандарт C++11 не утверждает этого относительно функций-членов `emplace_after()` и `push_front()`, но это, скорее всего, просто недоразумение.

Если использовать контейнер на основе узлов невозможно и в то же время необходимо реализовать принцип фиксации и отката, следует предусмотреть оболочки для каждой критичной операции. Например, следующая функция почти безопасно вставляет значение в указанную позицию любого контейнера:

```
template <typename T, typename Cont, typename Iter>
void insert (Cont& coll, const Iter& pos, const T& value)
{
   Cont tmp(coll);                // копируем контейнер и все элементы
   try {
      coll.insert(pos,value);     // пытаемся модифицировать копию copy
   }
   catch (...) {                  // в случае исключения
      coll.swap(tmp);             // - восстанавливаем исходный контейнер
      throw;                      // - и повторно генерируем исключение
   }
}
```

Следует обратить внимание на слово "почти", потому что эта функция не идеальна: операция `swap()` генерирует исключение, если операция копирования критерия сравнения для ассоциативного контейнера генерирует исключение. Как видим, обработка исключений — действительно нелегкое дело.

6.13. Расширение библиотеки STL

Библиотека STL представляет собой каркас, который можно расширять в любом направлении.

6.13.1. Интеграция дополнительных типов

Программист может создавать свои собственные контейнеры, итераторы, алгоритмы или функциональные объекты, при условии, что они удовлетворяют определенным требованиям. Собственно, в стандартной библиотеке C++ нет некоторых полезных расширений. Это сделано потому, что комитет остановил включение новых средств и сосредоточился на усовершенствовании существующих компонентов, иначе работа никогда не была бы закончена. По этой причине, например, хеш-таблицы не стали частью стандарта C++98.

В качестве полезных расширений можно назвать итераторы (разделы 9.6 и 14.3), контейнеры (раздел 7.10) и алгоритмы (разделы 7.6.2 и 9.5.1). Отметим, что все эти расширения следуют принципам обобщенного программирования.

- Все, что *ведет* себя как контейнер, *является* контейнером.
- Все, что *ведет* себя как итератор, *является* итератором.

Таким образом, работая с контейнерным классом, можно интегрировать его в библиотеку STL, снабдив соответствующим интерфейсом (`begin()`, `end()` и некоторыми определениями типов и т.д.). Если невозможно добавить функции-члены в такой класс, то можно создать класс-оболочку, обеспечивающий соответствующие итераторы.

Отметим, тем не менее, что некоторые контейнерные объекты не соответствуют концепции библиотеки STL. Например, тот факт, что контейнеры STL имеют начало и конец,

затрудняет создание кольцевых контейнеров, таких как кольцевой буфер, в рамках библиотеки STL.

В разделе 7.1.2 перечислены все общие контейнерные операции и отмечены те из них, которые требуются для контейнеров STL. Тем не менее это не значит, что соответствовать принципам библиотеки STL можно, лишь удовлетворив *всем* этим требованиям. Как правило, достаточно лишь частично выполнить эти требования. Даже некоторые стандартные контейнеры STL нарушают эти требования. Например, класс `forward_list` не имеет функции-члена `size()`, а класс `array` не соответствует общему требованию, что контейнер STL, инициализированный конструктором по умолчанию, должен быть пустым.

6.13.2. Наследование типов библиотеки STL

Другой вопрос — можно ли расширить возможности типов из библиотеки STL, создав производные классы и добавив в них функциональность. Обычно это невозможно. По причинам, связанным с производительностью, все классы библиотеки STL не содержат виртуальных функций, и поэтому не поддерживают полиморфизм посредством открытого наследования. Для того чтобы добавить в контейнер новую функциональность, необходимо определить новый класс, который внутренне использует классы библиотеки STL, или применить закрытое наследование.

Глава 7

Контейнеры STL

Эта глава, продолжающая обсуждение, начатое в главе 6, посвящена более глубокому описанию контейнеров STL. Она начинается с обзора общих возможностей и операций всех контейнерных классов и содержит подробное описание внутренней структуры, операций и производительности каждого контейнера. Кроме того, в ней показано, как использовать разнообразные операции, и приводятся нетривиальные примеры. Типичное использование каждого контейнера также иллюстрируется примерами. После этого обсуждается интересный вопрос: когда и какой контейнер использовать? В главе показано, как выбрать контейнер, наилучшим образом удовлетворяющий потребности программиста, сравнивая общие возможности, преимущества и недостатки всех контейнерных классов.

Эта глава тесно связана с главой 8, в которой подробно объясняются все члены, типы и операции контейнеров.

Стандартная библиотека C++ содержит особые контейнерные классы, так называемые *контейнерные адаптеры* (стек, очередь, очередь с приоритетами). Кроме того, некоторые классы обеспечивают контейнерный интерфейс (например, классы `string`, `bitset` и `valarray`). Все эти классы описываются отдельно[1]. Контейнерные адаптеры и битовые множества описываются в главе 12. Интерфейс строк библиотеки STL рассматривается в разделе 13.2.14. Класс `valarray` описывается в разделе 17.4.

7.1. Общие возможности и операции над контейнерами

7.1.1. Возможности контейнеров

В этом разделе описываются общие возможности контейнерных классов STL. Большая часть этих возможностей — требования, которые в принципе должен удовлетворять каждый контейнер STL. Перечислим три основных возможности.

1. Все контейнеры поддерживают семантику значений, а не ссылочную семантику. При вставке контейнеры копируют и/или перемещают элементы, а не работают со ссылками на них. Таким образом, теоретически каждый элемент контейнера STL должен допускать копирование и перемещение. Если объекты, которые необходимо сохранить в контейнере, не имеют открытого копирующего конструктора или их копирование затруднительно (например, потому, что оно занимает слишком много времени, или элементы могут принадлежать нескольким контейнерам

[1] Исторически сложилось так, что контейнерные адаптеры являются частью библиотеки. Однако с теоретической точки зрения они относятся не к архитектуре библиотеки STL, а только к ее использованию.

одновременно), к ним можно применять только операции перемещения или же хранить в контейнере указатели или объекты указателей, ссылающиеся на эти элементы. Пример использования разделяемых указателей для обеспечения ссылочной семантики описан в разделе 7.11.

2. Элементы в контейнере хранятся в определенном порядке. Каждый контейнерный тип имеет операции, возвращающие итераторы для перебора элементов. Это основной интерфейс для алгоритмов STL. Таким образом, если не выполняются операции вставки или удаления, при многократном переборе элементов порядок их следования не изменяется. Это относится даже к неупорядоченным контейнерам, если не выполняются операции добавления или удаления элементов или не инициируется внутренняя реорганизация контейнера.
3. В общем случае операции контейнеров не являются безопасными в том смысле, что они не проверяют все возможные ошибки. Вызывающая сторона должна гарантировать, что параметры операции соответствуют требованиям, установленным для этой операции. Нарушение этих требований, например некорректный индекс, приводит к неопределенному поведению, т.е. случиться может все, что угодно.
4. Обычно библиотека STL *не генерирует* исключений. Если пользовательские операции, вызванные контейнерами STL, генерируют исключение, ситуация изменяется. Подробности см. в разделе 6.12.2.

7.1.2. Операции над контейнерами

Стандарт содержит список общих требований, предъявляемых к контейнерам, которые применяются ко всем контейнерам STL. Однако вследствие разнообразия контейнеров, предусмотренных в стандарте C++11, допускаются исключения, позволяющие некоторым контейнерам не выполнять все требования, а также дополнительные операции, предусмотренные во всех контейнерах. В табл. 7.1 и 7.2 перечислены операции, общие для (почти) всех контейнеров. Столбец "Требование" содержит индикатор того, что операции являются частью общих требований, предъявляемых к контейнерам. Некоторые из этих операций рассматриваются в последующих подразделах.

Таблица 7.1. Общие операции (почти) всех контейнерных классов, часть 1

Операция	Требование	Действие
ContType c	Да	Конструктор по умолчанию; создает пустой контейнер, не содержащий ни одного элемента (контейнер `array<>` содержит элементы, заданные по умолчанию)
ContType c (c2)	Да	Копирующий конструктор; создает новый контейнер, являющийся копией контейнера *c2* (все элементы копируются)
ContType c = c2	Да	Копирующий конструктор; создает новый контейнер, являющийся копией контейнера *c2* (все элементы копируются)
ContType c (rv)	Да	Перемещающий конструктор; создает новый контейнер, получающий содержание от контейнера *rv* (в стандарте C++11; к классу `array<>` это не относится)

Окончание табл. 7.1

Операция	Требование	Действие
ContType c = rv	Да	Перемещающий конструктор; создает новый контейнер, получающий содержание от контейнера *rv* (в стандарте C++11; к классу `array<>` это не относится)
ContType c (beg, end)	–	Создает контейнер и инициализирует его копиями всех элементов интервала [*beg*,*end*) (к классу `array<>` это не относится)
ContType c (initlist)	–	Создает контейнер и инициализирует его копиями значения из списка инициализации *initlist* (в стандарте C++11; к классу `array<>` это не относится)
ContType c = initlist	–	Создает контейнер и инициализирует его копиями значения из списка инициализации *initlist* (в стандарте C++11)
c.~ContType ()	Да	Удаляет все элементы и освобождает память, если это возможно
c.empty()	Да	Возвращает признак того, что контейнер пуст (эквивалент `size()==0`, но может работать быстрее)
c.size()	Да	Возвращает текущее количество элементов (к классу `forward_list<>` это не относится)
c.max_size()	Да	Возвращает максимально возможное количество элементов
c1 == c2	Да	Возвращает признак того, что контейнер *c1* равен контейнеру *c2*
c1 != c2	Да	Возвращает признак того, что контейнер *c1* не равен контейнеру *c2* (эквивалент ! (*c1==c2*))
c1 < c2	–	Возвращает признак того, что контейнер *c1* меньше контейнера *c2* (это не относится к неупорядоченным контейнерам)
c1 > c2	–	Возвращает признак того, что контейнер *c1* больше контейнера *c2* (эквивалент *c2<c1*; это не относится к неупорядоченным контейнерам)
c1 <= c2	–	Возвращает признак того, что контейнер *c1* меньше или равен контейнеру *c2* (эквивалент ! (*c2<c1*); это не относится к неупорядоченным контейнерам)
c1 >= c2	–	Возвращает признак того, что контейнер *c1* больше или равен контейнеру *c2* (эквивалент ! (*c2<c1*); это не относится к неупорядоченным контейнерам)
c = c2	Да	Присваивает все элементы контейнера *c2* контейнеру *c*
c = rv	Да	Присваивает с перемещением все элементы контейнера *rv* контейнеру *c* (в стандарте C++11; это не относится к классу `array<>`)
c = initlist	–	Присваивает все элементы списка инициализации *initlist* (в стандарте C++11; это не относится к классу `array<>`)
c1.swap(c2)	Да	Обменивает данные контейнеров *c1* и *c2*
swap(c1,c2)	Да	Обменивает данные контейнеров *c1* и *c2*

Таблица 7.2. Общие операции (почти) всех контейнерных классов, часть 2

Операция	Требования	Действие
`c.begin()`	Да	Возвращает итератор, установленный на первый элемент
`c.end()`	Да	Возвращает итератор, установленный на позицию, следующую за последним элементом
`c.cbegin()`	Да	Возвращает константный итератор, установленный на первый элемент (в стандарте C++11)
`c.cend()`	Да	Возвращает константный итератор, установленный на позицию, следующую за последним элементом (в стандарте C++11)
`c.clear()`	–	Удаляет все элементы (опустошает контейнер; это не относится к классу `array<>`)

Инициализация

Каждый контейнерный класс имеет конструктор по умолчанию, копирующий конструктор и деструктор. Контейнер можно инициализировать элементами из заданного интервала и, по стандарту C++11, с помощью списка инициализации.

Конструктор для списка инициализации (см. раздел 3.1.3) обеспечивает удобный способ задания начальных значений. Это особенно полезно при инициализации константных контейнеров.

```
// инициализация вектора конкретными значениями (по стандарту C++11)
const std::vector<int> v1 = { 1, 2, 3, 5, 7, 11, 13, 17, 21 };

// то же самое с другим синтаксисом
const std::vector<int> v2 { 1, 2, 3, 5, 7, 11, 13, 17, 21 };

// инициализация неупорядоченного множества строкой "hello"
// и двумя пустыми строками
std::unordered_set<std::string> w = { "hello", std::string(), "" };
```

Использование списка инициализации для контейнеров класса `array<>` регламентируется отдельными правилами (см. раздел 7.2.1).

Конструктор для заданного интервала обеспечивает возможность инициализации контейнера элементами другого контейнера, массива в стиле языка C или из потока ввода. Этот конструктор является шаблонным членом (см. раздел 3.2), поэтому не только контейнер, но и тип его элементов могут отличаться от данного класса, при условии, что существует автоматическое преобразование из типа элементов контейнера-источника в тип элементов контейнера-приемника. Например, существуют следующие возможности.

- Можно инициализировать контейнер элементами другого контейнера.

```
std::list<int> l; // l – связанный список целых чисел
...
// копируем  в вектор все элементы списка как объекты типа floats
std::vector<float> c(l.begin(),l.end());
```

- По стандарту C++11 можно также переместить элементы, используя итератор перемещения (см. раздел 9.4.4).

```
std::list<std::string> l; // l – связанный список строк
...
// перемещаем все элементы списка в вектор
std::vector<std::string> c(std::make_move_iterator(l.begin()),
                           std::make_move_iterator(l.end()));
```

- Можно инициализировать контейнер элементами обычного массива в стиле языка C.

```
int carray[] = { 2, 3, 17, 33, 45, 77 };
...
// копируем в множество все элементы массива в стиле языка C
std::set<int> c(std::begin(carray),std::end(carray));
```

- Начиная со стандарта C++11 итераторы `std::begin()` и `std::end()` для массивов в стиле языка C определены в заголовочном файле `<iterator>`.
- Отметим, что до появления стандарта C++11 приходилось выполнять вызов

```
std::set<int> c(carray,carray+sizeof(carray)/sizeof(carray[0]));
```

- Можно инициализировать контейнер из потока ввода.

```
// считываем все целочисленные элементы в дека из потока ввода
std::deque<int> c(std::istream_iterator<int>(std::cin),
                  std::istream_iterator<int>());
```

- Отметим, что необходимо использовать новый единообразный синтаксис с фигурными скобками (см. раздел 3.1.3). В противном случае приходится использовать дополнительные круглые скобки, содержащие аргументы инициализатора.

```
// считываем все целочисленные элементы в дек из потока ввода
std::deque<int> c((std::istream_iterator<int>(std::cin)),
                  (std::istream_iterator<int>()));
```

- Причина заключается в том, что без дополнительной пары скобок аргумент имеет совершенно другой смысл, поэтому для следующих операторов компилятор выдаст странные предупреждения или сообщения об ошибке. Рассмотрим оператор без дополнительной пары скобок:

```
std::deque<int> c(std::istream_iterator<int>(std::cin),
                  std::istream_iterator<int>());
```

- В данном случае контейнер `c` объявляет *функцию*, возвращающую значение типа `deque<int>`. Ее первый параметр имеет тип `istream_iterator<int>` и имя `cin`, а второй безымянный параметр имеет тип "функции, не принимающей параметров и возвращающей объект типа `istream_iterator<int>`". Эта конструкция является вполне корректной с синтаксической точки зрения и как объявление, и как выражение. По правилам языка она интерпретируется как объявление. Дополнительная пара скобок вынуждает инициализатор не применять синтаксические правила объявления[2].

[2] Благодарю за эти разъяснения Джона Спайсера (John H. Spicer) из компании EDG.

В принципе эти приемы позволяют присваивать и вставлять элементы из другого интервала. Однако для этих операций точный интерфейс либо отличается наличием дополнительных аргументов, либо не предусматривается для всех контейнерных классов.

В заключение отметим, что стандарт C++11 позволяет использовать конструктор перемещения (см. раздел 3.1.5) для инициализации контейнера (для класса `array<>` он определяется неявно).

```
std::vector<int> v1;
...
// перемещаем содержимое контейнера v1 в контейнер v2,
// после чего состояние контейнера v1 становится неопределенным
std::vector<int> v2 = std::move(v1);
```

В результате вновь созданный контейнер содержит элементы контейнера, использованного для инициализации, а содержимое контейнера-источника становится неопределенным. Этот конструктор значительно повышает быстродействие программы, поскольку элементы перемещаются с помощью перестановки указателей, а не поэлементного копирования. Таким образом, если скопированный контейнер больше не нужен, следует использовать конструктор перемещения.

Присваивание и функция swap()

Во время присваивания контейнеров все элементы контейнера-источника копируются в контейнер-приемник, а старые элементы контейнера-приемника удаляются из него. Таким образом, присваивание контейнеров является относительно затратным.

По стандарту C++11 вместо этого можно использовать семантику перемещающего присваивания (см. раздел 3.1.5). Все контейнеры реализуют операцию перемещающего присваивания (причем класс `array<>` делает это неявно). При этом они просто переставляют указатели в памяти, а не копируют все значения. Точное поведение в этой ситуации не определено, но гарантированная константная сложность этой операции приводит к реализации, описанной ниже. Стандартная библиотека C++ просто обеспечивает, что после перемещающего присваивания контейнер, стоящий в левой части оператора присваивания, содержит элементы контейнера, стоящего в правой части этого оператора. После этого содержимое контейнера, стоящего в правой части операции, становится неопределенным.

```
std::vector<int> v1;
std::vector<int> v2;
...
// перемещаем содержимое контейнера v1 в контейнер v2,
// после этого состояние контейнера v1 становится неопределенным
v2 = std::move(v1);
```

Если после присваивания содержимое контейнера, стоящего в правой части оператора, больше не используется, то для повышения быстродействия следует применять перемещающее присваивание.

По стандарту C++98 все контейнеры имеют функцию-член `swap()` для обмена содержимого двух контейнеров. Фактически эта операция сводится к простой перестановке внутренних указателей, ссылающихся на данные (элементы, распределители, критерий сортировки). Таким образом, функция `swap()` гарантированно обеспечивает константную, а не линейную сложность копирующего присваивания. Итераторы и ссылки на элементы контейнера следуют за переставляемыми элементами. Следовательно, после выполнения

функции swap() итераторы и ссылки по-прежнему будут ссылаться на элементы, на которые они ссылались раньше, однако в другом контейнере.

Отметим, что с контейнерами типа array<> функция swap() работает несколько иначе. Поскольку мы не можем просто переставить внутренние указатели, функция swap() имеет линейную сложность, а после ее выполнения итераторы и ссылки ссылаются на тот же контейнер, но на другие элементы.

Операции над размерами

Почти для всех контейнерных классов предусмотрены три операции над размерами.

1. Функция **empty()** возвращает признак того, что количество элементов равно нулю (begin()==end()). Эту функцию целесообразно применять вместо оператора size()==0, потому что она реализована более эффективно, чем функция size(), причем функция size() не работает с последовательными списками.
2. Функция **size()** возвращает текущее количество элементов контейнера. Эта операция не поддерживается классом forward_list<>, потому что в этом случае она не имела бы константной сложности.
3. Функция **max_size()** возвращает максимальное количество элементов, которое может содержать контейнер. Это значение зависит от реализации. Например, вектор обычно содержит все элементы в отдельном блоке памяти, поэтому могут возникнуть ограничения, связанные с организацией памяти в компьютере. Если таких ограничений нет, функция max_size() обычно возвращает максимальное значение для типа индекса.

Сравнения

Во всех контейнерах, за исключением неупорядоченных, определены обычные операторы сравнения ==, !=, <, <=, > и >=, подчиняющиеся трем правилам.

1. Оба контейнера должны быть однотипными.
2. Два контейнера считаются равными, если их элементы равны и располагаются в одинаковом порядке. Для проверки равенства элементов используется оператор ==.
3. Для проверки того, какой из двух контейнеров меньше использует лексикографическое сравнение (см. раздел 11.5.4).

Для неупорядоченных контейнеров определены только операции == и !=. Они возвращают значение true, если каждому элементу в одном контейнере соответствует равный ему элемент в другом контейнере. В этом случае порядок значения не имеет (именно поэтому эти контейнеры называются неупорядоченными).

Поскольку операции <, <=, > и >= не определены для неупорядоченных контейнеров, общим требованием к контейнерам является предоставление только операторов == и !=. До появления стандарта C++11 требовались все операторы сравнения. В стандарте C++11 имеется таблица необязательных требований к контейнерам, в которую включены остальные четыре операции сравнения.

Для сравнения контейнеров разных типов следует использовать алгоритмы сравнения, описанные в разделе 11.5.4.

Доступ к элементам

Все контейнеры реализуют интерфейс итераторов, т.е. поддерживают диапазонный цикл `for` (см. раздел 3.1.4). Таким образом, в соответствии со стандартом C++11 проще всего получить доступ ко всем элементам следующим образом:

```
for (const auto& elem : coll) {
     std::cout << elem << std::endl;
}
```

Для того чтобы иметь возможность манипулировать элементами, следует пропустить ключевое слово `const`:

```
for (auto& elem : coll) {
     elem = ...;
}
```

Для того чтобы оперировать позициями (например, иметь возможность вставлять, удалять и перемещать элементы), всегда можно использовать итераторы, возвращаемые функциями `cbegin()` и `cend()` и предназначенные только для чтения:

```
for (auto pos=coll.cbegin(); pos!=coll.cend(); ++pos) {
     std::cout << *pos << std::endl;
}
```

и итераторы, возвращаемые функциями `begin()` и `end()`, и предназначенные для чтения и для записи:

```
for (auto pos=coll.begin(); pos!=coll.end(); ++pos) {
     *pos = ...;
}
```

До появления стандарта C++11 программист был обязан, а теперь просто может явно объявить тип итератора только для чтения:

```
colltype::const_iterator pos;
for (pos=coll.begin(); pos!=coll.end(); ++pos) {
     ...;
}
```

или для записи:

```
colltype::iterator pos;
for (pos=coll.begin(); pos!=coll.end(); ++pos) {
     ...;
}
```

Все контейнеры, за исключением векторов и деков, гарантируют, что итераторы и ссылки на элементы остаются корректными после удаления любых остальных элементов. Для векторов корректными остаются только элементы, расположенные до точки удаления.

После удаления всех элементов с помощью функции `clear()` в векторах, деках и строках *запредельный итератор*, возвращенный функциями `end()` и `cend()`, может стать некорректным.

После вставки элементов только списки, односвязные списки и ассоциативные контейнеры гарантируют, что итераторы и ссылки на элементы останутся корректными. Для векторов эта гарантия имеет место, только если вставки не превышают емкость контейнера. Для неупорядоченных контейнеров эта гарантия распространяется в основном на ссылки, а для итераторов она действует, только если не выполнялось повторное хеширование, т.е. количество вставленных элементов не превышает произведения количества "корзин" на максимальный коэффициент загрузки.

7.1.3.Типы контейнеров

Все контейнеры предоставляют определения общих типов, перечисленных в табл. 7.3.

Таблица 7.3. Общие типы, определенные во всех контейнерных классах

Тип	Требование	Действие
`size_type`	Да	Целочисленный тип без знака для значений размера
`difference_type`	Да	Целочисленный тип со знаком для разности между значениями
`value_type`	Да	Тип элементов
`reference`	Да	Тип ссылок на элементы
`const_reference`	Да	Тип константных ссылок на элементы
`iterator`	Да	Тип итераторов
`const_iterator`	Да	Тип итераторов только для чтения
`pointer`	–	Тип указателей на элементы (по стандарту C++11)
`const_pointer`	–	Тип указателей на элементы, предназначенные только для чтения (по стандарту C++11)

7.2. Массивы

Массив — это экземпляр контейнерного класса `array<>`, моделирующий статический массив. Он содержит обычный статический массив в стиле языка C и предоставляет интерфейс контейнера STL (рис. 7.1). С теоретической точки зрения массив — это последовательность элементов, имеющая постоянный размер. Таким образом, невозможно ни добавить, ни удалить элементы, чтобы изменить размер массива. Можно лишь заменять элементы массива.

Рис. 7.1. Структура массива

Класс array<>, введенный в стандартную библиотеку C++ в соответствии с документом TR1, стал "наследником" полезного класса-оболочки для обычных массивов в стиле языка C, описанного Бъярне Страуструпом (Bjarne Stroustrup) в книге [*Stroustrup:C++*]. Он является более безопасным и не менее быстродействующим, чем обычные массивы.

Для использования массива необходимо включить в программу заголовочный файл <array>:

```
#include <array>
```

где тип определен как шаблонный класс в пространстве имен std:

```
namespace std {
   template <typename T, size_t N>
   class array;
}
```

Элементы массива могут иметь любой тип T.

Второй шаблонный параметр задает количество элементов массива на протяжении его существования. Таким образом, функция size() всегда возвращает N.

Распределитель памяти не предусмотрен.

7.2.1. Возможности массивов

Массивы копируют свои элементы во внутренний статический массив в стиле языка C. Элементы всегда располагаются в определенном порядке. Таким образом, массивы являются разновидностью *упорядоченной коллекции*. Массивы обеспечивают *прямой доступ*. Таким образом, программист может получить прямой доступ к каждому элементу массива за константное время, при условии, что он знает его позицию. Итераторы массивов являются итераторами произвольного доступа, поэтому можно использовать любой алгоритм из библиотеки STL.

Если возникает необходимость работать с последовательностью, состоящей из фиксированного количества элементов, класс array<> обеспечивает наилучшую производительность, потому что память выделяется в стеке (по возможности), повторное выделение памяти никогда не происходит и программист имеет прямой доступ к элементам массива.

Инициализация

Инициализация объектов класса array<> имеет уникальную семантику. В качестве первой особенности укажем, что конструктор по умолчанию не создает пустой контейнер, потому что количество элементов в контейнере всегда является постоянным и равным второму шаблонному параметру на всем протяжении существования объекта.

Отметим, что класс array<> — это единственный контейнер, элементы которого инициализируются по умолчанию, если для их инициализации ничего не передается. Это значит, что для основных типов начальное значение может оказаться не нулем, а чем-то неопределенным (см. раздел 3.2.1). Например:

```
std::array<int,4> x; // Ой!: элементы массива x имеют неопределенные значения
```

Вместо этого можно предусмотреть пустой инициализатор. В этом случае все значения обязательно будут инициализированы, причем элементы основных типов будут *инициализированы нулем*:

```
std::array<int,4> x = {}; // OK: все элементы массива x равны 0 (int())
```

Причина заключается в том, что, хотя класс `array<>` на первый взгляд содержит конструктор для списка инициализации, на самом деле это не так. Вместо этого класс `array<>` подчиняется требованиям, предъявляемым к агрегатам[3]. Следовательно, еще до появления стандарта C++11 можно было использовать список инициализации массива при его создании:

```
std::array<int,5> coll = { 42, 377, 611, 21, 44 };
```

Элементы списка инициализации должны иметь один и тот же тип или предусматривать преобразование в тип элемента массива.

Если список инициализации содержит недостаточное количество элементов, элементы массива инициализируются конструктором, предусмотренным по умолчанию для типа элемента. В данном случае гарантируется, что данные основных типов инициализируются нулями. Например:

```
std::array<int,10> c2 = { 42 }; // один элемент со значением 42,
                                // за которым следуют 9 элементов
                                // со значением 0
```

Если количество элементов в списках инициализации больше, чем размер массива, выражение считается некорректным.

```
std::array<int,5> c3 = { 1, 2, 3, 4, 5, 6 }; // ОШИБКА: слишком много значений
```

Поскольку для списков инициализации не предусмотрены конструкторы и операторы присваивания, инициализация массива во время его объявления является единственным вариантом использования списков инициализации. По этой причине нельзя также использовать скобки для задания начальных значений (в отличие от других контейнерных типов).

```
std::array<int,5> a({ 1, 2, 3, 4, 5, 6 }); // ОШИБКА


std::vector<int> v({ 1, 2, 3, 4, 5, 6 });  // OK
```

То, что класс `array<>` является агрегатом, означает также, что члены, содержащие все элементы, являются открытыми. Однако их имена в стандарте не перечислены; таким образом, любое прямое обращение к открытому члену, содержащему все элементы, приводит к непредсказуемым последствиям и определенно не является переносимым.

Функция swap() и семантика перемещения

Как и любой другой контейнер, класс `array<>` предоставляет операцию `swap()`. Таким образом, элементы контейнеров одинаковых типов можно менять местами (элементы

[3] Агрегат — это массив или класс, не имеющий пользовательских конструкторов, закрытых или защищенных нестатических данных-членов, базовых классов и виртуальных функций.

должны иметь одинаковые типы, и количество элементов в контейнерах должно быть одинаковым). Отметим, однако, что класс `array<>` не может просто поменять местами внутренние указатели. По этой причине функция `swap()` имеет линейную сложность, а итераторы и ссылки не обмениваются между контейнерами вместе со своими элементами. Итак, после обмена итераторы и ссылки относятся к тому же самому контейнеру, но ссылаются на другие элементы.

К массивам можно применить семантику перемещения. Например[4]:

```
std::array<std::string,10> as1, as2;
...
as1 = std::move(as2);
```

Размер

Размер массива, не содержащего ни одного элемента, можно задать равным `0`. В этом случае функции `begin()` и `end()`, `cbegin()` и `cend()`, а также соответствующие обратные итераторы будут возвращать одно и то же уникальное значение. Однако значения, возвращаемые функциями `front()` и `back()`, являются неопределенными.

```
std::array<Elem,0> coll;              // массив без элементов
std::sort(coll.begin(),coll.end());   // OK (но не имеет последствий)
coll[5] = elem;                       // ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ:
                                      // непредсказуемые последствия
std::cout << coll.front();            // ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ:
                                      // непредсказуемые последствия
```

Значение, возвращаемое функцией `data()`, остается незаданным, т.е. можно передавать возвращаемое значение в другие места, лишь бы не выполнялось его разыменование.

7.2.2. Операции над массивами

Создание, копирование и уничтожение

В табл. 7.4 перечислены конструкторы и деструкторы массивов. Поскольку класс `array<>` является агрегатом, эти операции определяются только неявно. Можно создавать массивы как с элементами для инициализации, так и без них. Конструктор, заданный по умолчанию, инициализирует элементы по умолчанию, т.е. значения основных типов остаются неопределенными. Если список инициализации содержит недостаточное количество элементов, остальные элементы создаются их конструктором, заданным по умолчанию (и `0` для основных типов). Замечания о возможных источниках инициализации см. в разделе 7.1.2.

Отметим, что в отличие от других контейнеров использование круглых скобок в списках инициализаторов невозможно:

```
std::array<int,5> a(( 1, 2, 3, 4, 5 )); // ОШИБКА
```

[4] Благодарю за этот пример Дэниэля Крюглера (Daniel Krügler).

Таблица 7.4. Конструкторы и деструктор класса `array<>`

Операция	Действие
`array<Elem,N> c`	Конструктор по умолчанию; создает массив с элементами, инициализированными по умолчанию
`array<Elem,N> c(c2)`	Копирующий конструктор; создает копию другого массива того же типа (все элементы копируются)
`array<Elem,N> c = c2`	Копирующий конструктор; создает копию другого массива того же типа (все элементы копируются)
`array<Elem,N> c(rv)`	Перемещающий конструктор; создает новый массив, перемещая (или копируя) элементы массива *rv* (по стандарту C++11)
`array<Elem,N> c = rv`	Перемещающий конструктор; создает новый массив, перемещая (или копируя) массива *rv* (по стандарту C++11)
`array<Elem,N> c = initlist`	Создает массив, инициализированный элементами из списка инициализации
`c.~array()`	Уничтожает все элементы

Немодифицирующие операции

В табл. 7.5 перечислены все немодифицирующие операции над массивами. Дополнительные замечания приведены в разделе 7.1.2.

Таблица 7.5. Немодифицирующие операции класса `array<>`

Операция	Действие
`c.empty()`	Возвращает признак того, что контейнер пуст (эквивалент `size()==0`, но может работать быстрее)
`c.size()`	Возвращает текущее количество элементов
`c.max_size()`	Возвращает максимально возможное количество элементов
c1 == *c2*	Проверка того, что массив *c1* равен массиву *c2* (выполняет оператор == для всех элементов)
c1 != *c2*	Проверка того, что массив *c1* не равен массиву *c2* (эквивалент `!(c1==c2)`)
c1 < *c2*	Проверка того, что массив *c1* меньше массива *c2*
c1 > *c2*	Проверка того, что массив *c1* больше массива *c2* (эквивалент *c2*<*c1*)
c1 <= *c2*	Проверка того, что массив *c1* меньше или равен массиву *c2* (эквивалент `!(c2<c1)`)
c1 >= *c2*	Проверка того, что массив *c1* больше или равен массиву *c2* (эквивалент `!(c1<c2)`)

Присваивание

В табл. 7.6 перечислены способы присваивания новых значений. Помимо операции присваивания, для присваивания значения каждому элементу можно использовать только

функцию `fill()`, а для обмена элементами между массивами — функцию `swap()`. Для успешного выполнения операции = и вызова функции `swap()` оба массива должны иметь один и тот же тип, т.е. типы элементов и размеры массивов должны совпадать.

Таблица 7.6. Операции присваивания класса `array<>`

Операция	Действие
c = c2	Присваивает все элементы массива *c2* массиву *c*
c = *rv*	Перемещает все элементы массива *rv* в массив *c* (по стандарту C++11)
c.fill(*val*)	Присваивает значение *val* каждому элементу массива *c*
c1.swap(*c2*)	Обменивает элементы массивов *c1* и *c2*
swap(*c1*,*c2*)	Обменивает элементы массивов *c1* и *c2*

Отметим, что функция `swap()` для массивов не гарантирует константную сложность, поскольку обменивать внутренние указатели невозможно (см. раздел 7.2.1). Вместо этого, как и в алгоритме `swap_ranges()` (см. раздел 11.6.4), все элементы обоих массивов получают новые значения.

Внутренне все эти операции вызывают оператор присваивания, предусмотренный для типа элемента.

Доступ к элементам

Для доступа ко всем элементам массива необходимо использовать диапазонный цикл `for` (см. раздел 3.1.4), специальные операции или итераторы. Кроме того, предусмотрен интерфейс кортежа, поэтому для доступа к конкретному элементу можно использовать функцию `get<>()` (см. раздел 7.2.5). В табл. 7.7 перечислены все операции для непосредственного доступа к элементам массива. Как принято в языках C и C++, первый элемент имеет индекс `0`, а последний — `size()-1`. Таким образом, n-й элемент имеет индекс n−1. Для неконстантных массивов эти операции возвращают ссылку на элемент. Следовательно, с помощью этих операций можно модифицировать элементы, если это не запрещено по другим причинам.

Таблица 7.7. Операции непосредственного доступа к элементам объекта класса `array<>`

Операция	Действие
c[*idx*]	Возвращает элемент с индексом *idx* (проверка выхода за пределы допустимого диапазона *не выполняется*)
c.at(*idx*)	Возвращает элемент с индексом *idx* (при выходе за пределы допустимого диапазона *генерирует исключение*)
c.front()	Возвращает первый элемент (проверка существования первого элемента *не выполняется*)
c.back()	Возвращает последний элемент (проверка существования последнего элемента *не выполняется*)

Самый важный момент для вызывающей стороны состоит в том, выполняют ли эти операции проверку выхода за пределы допустимого диапазона. Эту проверку выполняет только функция `at()`. Если индекс выходит за пределы допустимого диапазона, функция `at()`

генерирует исключение out_of_range (см. раздел 4.3). Все остальные функции проверку *не производят*. Выход за пределы допустимого диапазона приводит к непредсказуемым последствиям. Выполнение операции [ ] и вызов функции front() и back() для пустого объекта класса array<> всегда приводит к непредсказуемым последствиям. Однако следует подчеркнуть, что массив считается пустым, только если его размер равен 0.

```
std::array<Elem,4> coll;     // только четыре элемента!
coll[5] = elem;              // ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ⇒
                             // непредсказуемые последствия
std::cout << coll.front();   // OK (массив coll после создания имеет 4 элемента)
std::array<Elem,0> coll2;    // всегда пустой
std::cout << coll2.front(); // ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ⇒
                             // непредсказуемые последствия
```

Итак, в сомнительных ситуациях необходимо убедиться, что индекс в операции [ ] является корректным, или использовать функцию at().

```
template <typename C>
void foo (C& coll)
{
   if (coll.size() > 5) {
      coll[5] = ...; // OK
   }

   coll.at(5) = ...; // генерирует исключение out_of_range
}
```

Отметим, что этот код является корректным только в однопоточной среде. В многопотоковом контексте необходим механизм синхронизации, предотвращающий модификацию массива coll между проверкой его размера и доступом к его элементам (см. раздел 18.4.3).

Функции для итераторов

Массивы предоставляют обычные операции для получения итераторов (табл. 7.8). Итераторы массивов являются итераторами произвольного доступа (см. раздел 9.2). Следовательно, при работе с массивами можно использовать все алгоритмы STL.

Точный тип этих итераторов определяется реализацией. Однако для массивов итераторы, возвращаемые функциями begin(), cbegin(), end() и cend(), часто являются обычными указателями, и это отлично, потому что в классе array<> скрывается массив в стиле языка C, для которого элементы и обычные указатели образуют интерфейс итераторов произвольного доступа. Однако то, что итераторы представляют собой обычные указатели, использовать невозможно. Например, при работе с безопасной версией библиотеки STL, проверяющей ошибки выхода за пределы допустимого диапазона и другие потенциальные проблемы, тип итератора обычно представляет собой вспомогательный класс. В разделе 9.2.6 продемонстрирована огромная разница между итераторами, реализованными как указатели, и итераторами, реализованными как классы.

Итераторы остаются корректными, пока массив остается корректным. Однако, в отличие от любых других контейнеров, при работе с массивом функция swap() присваивает новые значения элементам, на которые ссылаются итераторы, ссылки и указатели.

Таблица 7.8. Операции над итераторами класса array<>

Операция	Действие
c.begin()	Возвращает итератор произвольного доступа, установленный на первый элемент
c.end()	Возвращает итератор произвольного доступа, установленный на позицию, следующую за последним элементом
c.cbegin()	Возвращает константный итератор произвольного доступа, установленный на первый элемент (по стандарту C++11)
c.cend()	Возвращает константный итератор произвольного доступа, установленный на позицию, следующую за последним элементом (начиная со стандарта C++11)
c.rbegin()	Возвращает обратный итератор, установленный на первый элемент в обратном обходе
c.rend()	Возвращает обратный итератор, установленный на позицию, следующую за последним элементом в обратном обходе
c.crbegin()	Возвращает константный обратный итератор, установленный на первый элемент в обратном обходе (по стандарту C++11)
c.crend()	Возвращает константный обратный итератор, установленный на позицию, следующую за последним элементом в обратном обходе (по стандарту C++11)

7.2.3. Использование объектов array<> как массивов в стиле языка C

Аналогично классу vector<>, стандартная библиотека C++ гарантирует, что элементы объекта класса array<> занимают соседние ячейки памяти. Таким образом, можно ожидать, что для любого корректного индекса i в массиве a следующее выражение будет равно значению true:

```
&a[i] == &a[0] + i
```

Эта гарантия имеет важные последствия. Она просто означает, что можно использовать класс array<> всюду, где можно использовать обычный массив в стиле языка C. Например, можно использовать массив для хранения обычных строк в стиле языка C типа char* или const char*.

```
std::array<char,41> a;          // создаем статический массив из 41 символа
strcpy(&a[0],"hello, world");   // копируем в массив строку в стиле языка C
printf("%s\n", &a[0]);          // выводим на экран содержимое массива как
                                // строку в стиле языка C
```

Однако следует подчеркнуть, что выражение &a[0] не следует использовать для непосредственного доступа к элементам массива, потому что для этой цели предназначена специальная функция-член data().

```
std::array<char,41> a;            // создаем статический массив из 41 символа
strcpy(a.data(),"hello, world");  // копируем в массив строку в стиле языка C
printf("%s\n", a.data());         // выводим на экран содержимое массива как
                                  // строку в стиле языка C
```

Разумеется, следует быть осторожным при таком использовании объектов класса array<> (точно так же, как при работе с обычными массивами и указателями в стиле

языка C). Например, необходимо убедиться, что размер массива достаточно велик, чтобы в него можно было копировать данные, а если массив содержит строки в стиле языка C, то в его конце записан элемент '\0'. Однако этот пример показывает, что если программисту по каким-то причинам понадобится массив типа T, например, для существующей библиотеки языка C, то там, где требуется интерфейс в стиле языка C, можно использовать класс array<> (или vector<>) и функцию data().

Отметим, что вы не должны передавать итератор как адрес первого элемента. Тип итераторов класса array<> зависит от реализации, и он может резко отличаться от обычного указателя.

```
printf("%s\n", a.begin()); // ОШИБКА (работоспособно, но не машинонезависимо)
printf("%s\n", a.data());  // ОК
```

7.2.4. Обработка исключений

Массивы обеспечивают минимальную поддержку проверки логических ошибок. Единственной функцией-членом, которой стандарт разрешает генерировать исключение, является функция at(), представляющая собой безопасный вариант операции индексирования (см. раздел 7.2.2).

На функции, вызываемые массивом (функций типа элементов или пользовательских функций), не распространяется обычная гарантия (поскольку невозможно вставлять или удалять элементы, исключение может возникнуть только в том случае, когда происходит копирование, перемещение или присваивание). Особо следует отметить, что функция swap() может генерировать исключение, потому что она производит поэлементный обмен, который может порождать исключения.

Общие вопросы, связанные с обработкой исключений в библиотеке STL, см. в разделе 6.12.2.

7.2.5. Интерфейс кортежа

Массивы обеспечивают интерфейс кортежа (см. раздел 5.1.2). Следовательно, можно использовать выражения tuple_size<>::value для определения количества элементов, tuple_element<>::type для выяснения типа конкретного элемента и get() для доступа к конкретному элементу. Например:

```
typedef std::array<std::string,5> FiveStrings;

FiveStrings a = { "hello", "nico", "how", "are", "you" };

std::tuple_size<FiveStrings>::value;          // возвращает 5
std::tuple_element<1,FiveStrings>::type;      // возвращает std::string
std::get<1>(a);                               // возвращает std::string("nico")
```

7.2.6. Примеры использования массивов

В следующем примере иллюстрируется простое использование объекта класса array<>:

```
// cont/array1.cpp

#include <array>
```

```
#include <algorithm>
#include <functional>
#include <numeric>
#include "print.hpp"
using namespace std;

int main()
{
   // создаем массив из 10 целых чисел
   array<int,10> a = { 11, 22, 33, 44 };

   PRINT_ELEMENTS(a);

   // изменяем два последних элемента
   a.back() = 9999999;
   a[a.size()-2] = 42;
   PRINT_ELEMENTS(a);

   // вычисляем сумму всех элементов
   cout << "sum: "
        << accumulate(a.begin(),a.end(),0)
        << endl;

   // меняем знак всех элементов на противоположный
   transform(a.begin(),a.end(),          // источник
                       a.begin(),        // назначение
                       negate<int>());   // операция
   PRINT_ELEMENTS(a);
}
```

Как видим, для непосредственного манипулирования контейнером можно использовать общие операции контейнерного интерфейса (оператор =, функцию `size()` и операцию `[ ]`). Поскольку для доступа к итераторам используются такие функции-члены, как `begin()` и `end()`, можно также выполнять другие операции, которые вызывают функции `begin()` и `end()`, например, модифицирующие и немодифицирующие алгоритмы, а также вспомогательную функцию `PRINT_ELEMENTS()`, введенную в разделе 6.6.

Результат работы этой программы выглядит следующим образом:

```
11 22 33 44 0 0 0 0 0 0
11 22 33 44 0 0 0 0 42 9999999
sum: 10000151
-11 -22 -33 -44 0 0 0 0 -42 -9999999
```

7.3. Векторы

Вектор моделирует динамический массив. Таким образом, вектор — это абстракция, управляющая своими элементами в стиле динамического массива из языка C (рис. 7.2). Однако в стандарте не указано, что реализация вектора использует динамический массив. Скорее это следует из ограничений и спецификаций сложности его операций.

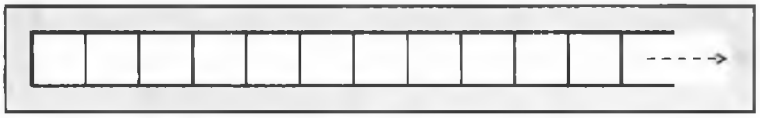


Рис. 7.2. Структура вектора

Для использования вектора необходимо включить заголовочный файл `<vector>`.

```
#include <vector>
```

В нем вектор определен как шаблонный класс в пространстве имен `std`.

```
namespace std {
   template <typename T,
             typename Allocator = allocator<T> >
   class vector;
}
```

Элементы вектора могут иметь любой тип `T`. Необязательный второй шаблонный параметр задает модель памяти (см. главу 19). По умолчанию в качестве модели памяти используется класс `allocator` из стандартной библиотеки C++.

7.3.1. Возможности векторов

Вектор копирует свои элементы во внутренний динамический массив. Его элементы всегда располагаются в определенном порядке. Таким образом, вектор — это разновидность *упорядоченной коллекции*. Вектор обеспечивает *прямой доступ*, т.е. программист имеет возможность использовать прямой доступ к каждому элементу за константное время, при условии, что он знает его позицию. Итераторы вектора являются итераторами произвольного доступа, поэтому их может использовать любой алгоритм STL.

Векторы обеспечивают хорошую производительность, если вставка или удаление элементов происходит в конце. При вставке или удалении в середине вектора или в его начале производительность ухудшается. Это объясняется тем, что элемент, расположенный за вставляемым или удаляемым элементом, перемещается на другую позицию. Фактически к каждому из последующих элементов должна применяться операция присваивания.

Размер и емкость

Один из приемов, благодаря которым векторы имеют хорошую производительность, заключается в выделении большего объема памяти, чем требуется для хранения всех его элементов. Для эффективного и правильного использования векторов необходимо понимать, как размер и емкость вектора совместно используются вектором.

Векторы имеют обычные операции для работы с размером: `size()`, `empty()` и `max_size()` (см. раздел 7.1.2). Дополнительная операция, связанная с размером, — функция `capacity()`, возвращающая количество элементов, которые вектор может хранить в памяти в данный момент. Если размер превышает значение, возвращаемое функцией `capacity()`, вектор повторно выделяет внутреннюю память.

Емкость вектора имеет значение по двум причинам.

1. Повторное выделение памяти делает некорректными все ссылки, указатели и итераторы, установленные на элементы вектора.
2. Повторное выделение памяти требует времени.

Таким образом, если программа работает с указателями, ссылками или итераторами внутри вектора или если целью является быстродействие, следует учитывать емкость вектора.

Для того чтобы избежать повторного выделения памяти, можно использовать функцию `reserve()`, гарантирующую определенную емкость еще до того, как она действительно потребуется. Таким образом вы можете гарантировать, что ссылки остаются корректными до тех пор, пока количество элементов не превысит емкость вектора.

```
std::vector<int> v;  // создаем пустой вектор
v.reserve(80);       // резервируем память для 80 элементов
```

Другой способ избежать повторного выделения памяти — инициализировать вектор достаточным количеством элементов, передав конструктору дополнительные аргументы. Например, если передать числовое значение как параметр, оно будет использовано как начальный размер вектора.

```
std::vector<T> v(5); // создаем вектор и инициализируем его пятью значениями
                     // (пять раз вызываем для типа T
                     // конструктор по умолчанию)
```

Разумеется, для этого тип элементов должен иметь конструктор по умолчанию. Для основных типов гарантируется инициализация нулями (см. раздел 3.2.1). Отметим, однако, что для сложных типов, даже если в них предусмотреть конструктор по умолчанию, инициализация требует времени. Если единственной причиной инициализации является резервирование памяти, следует применять функцию `reserve()`.

Концепция емкости векторов аналогична строкам (см. раздел 13.2.5), за одним важным исключением: в отличие от строк, невозможно вызвать функцию `reserve()` для уменьшения емкости вектора. Вызов функции `reserve()` с аргументом, который меньше, чем текущая емкость, считается фиктивной операцией.

Более того, способы достижения оптимальной производительности и эффективности использования памяти зависят от реализации. Таким образом, реализации могут увеличивать емкость более крупномасштабно. На самом деле, для того чтобы избежать внутренней фрагментации, многие реализации выделяют целый блок памяти (например, два килобайта) при выполнении первой вставки, если программист предварительно сам не вызвал функцию `reserve()`. При работе с многочисленными векторами, содержащими небольшое количество элементов, это может привести к избыточным затратам памяти.

Поскольку емкость векторов никогда не уменьшается, ссылки, указатели и итераторы при удалении элементов всегда остаются корректными, при условии, что они ссылаются на позицию, расположенную до обрабатываемых элементов. Однако при превышении емкости вектора вставки делают некорректными все ссылки, указатели и итераторы.

В языке C++11 появилась новая функция-член класса `vector`: запрос без привязки для уменьшения емкости до текущего количества элементов.

```
v.shrink_to_fit(); // запрос на уменьшение объема памяти (по стандарту C++11)
```

Этот запрос не имеет привязки, чтобы обеспечить свободу оптимизации во время реализации класса. Таким образом, не следует ожидать, что после выполнения запроса оператор `v.capacity()==v.size()` вернет значение `true`.

До стандарта C++11 уменьшить емкость вектора можно было только косвенно, обменивая содержимое с другим вектором. Следующая функция уменьшает емкость, сохраняя элементы:

```
template <typename T>
void shrinkCapacity(std::vector<T>& v)
{
   std::vector<T> tmp(v); // копируем элементы в новый вектор
   v.swap(tmp);           // обмениваем внутренние данные векторов
}
```

Уменьшить емкость можно и без вызова этой функции, выполнив следующий код[5]:

```
// уменьшаем емкость вектора v для типа T
std::vector<T>(v).swap(v);
```

Однако следует отметить, что после выполнения функции `swap()` все ссылки, указатели и итераторы также поменяют контейнеры. Они по-прежнему будут ссылаться на те элементы, на которые ссылались ранее. Таким образом, функция `shrinkCapacity()` делает некорректными все ссылки, указатели и итераторы. Это же относится и к функции `shrink_to_fit()`.

7.3.2. Операции над векторами

Создание, копирование и удаление

В табл. 7.9 перечислены конструкторы и деструкторы векторов. Векторы можно создавать как с элементами для инициализации, так и без них. Если передать только размер, то элементы будут созданы с помощью их конструкторов, заданных по умолчанию. Отметим, что явный вызов конструктора по умолчанию также инициализирует нулями основные типы, такие как `int` (см. раздел 3.2.1). Возможные источники инициализации рассматриваются в разделе 7.1.2.

Таблица 7.9. Конструкторы и деструкторы векторов

Операция	Действие
`vector<Elem>` *c*	Конструктор по умолчанию; создает пустой вектор, не содержащий элементы
`vector<Elem>` *c*(*c2*)	Копирующий конструктор; создает новый вектор как копию вектора *c2* (все элементы копируются)
`vector<Elem>` *c* = *c2*	Копирующий конструктор; создает новый вектор как копию вектора *c2* (все элементы копируются)

[5] Вы (или компилятор) могли бы рассматривать этот код как некорректный, потому что он вызывает неконстантную функию-член для временного значения. Однако стандарт C++ позволяет вызывать неконстантные функции-члены для временных значений.

Окончание табл. 7.9

Операция	Действие
`vector<Elem>` *c* (*rv*)	Перемещающий конструктор; создает новый вектор, принимающий содержимое вектора *rv* (по стандарту C++11)
`vector<Elem>` *c* = *rv*	Перемещающий конструктор; создает новый вектор, принимающий содержимое вектора *rv* (по стандарту C++11)
`vector<Elem>` *c* (*n*)	Создает вектор из *n* элементов с помощью конструктора по умолчанию
`vector<Elem>` *c* (*n*, *elem*)	Создает вектор, инициализированный *n* копиями элемента *elem*
`vector<Elem>` *c* (*beg*, *end*)	Создает вектор, инициализированный элементами из интервала [*beg*, *end*)
`vector<Elem>` *c* (*initlist*)	Создает вектор, инициализированный элементами списка инициализации *initlist* (по стандарту C++11)
`vector<Elem>` *c* = *initlist*	Создает вектор, инициализированный элементами списка инициализации `initlist` (по стандарту C++11)
c.~`vector()`	Уничтожает все элементы и освобождает память

Немодифицирующие операции

В табл. 7.10 перечислены все немодифицирующие операции над векторами[6]. Дополнительные замечания приведены в разделах 7.1.2 и 7.3.1.

Таблица 7.10. Немодифицирующие операции над векторами

Операция	Действие
c.`empty()`	Возвращает результат проверки контейнера на пустоту (эквивалент `size()==0`, но может работать быстрее)
c.`size()`	Возвращает текущее количество элементов
c.`max_size()`	Возвращает максимально возможное количество элементов
c.`capacity()`	Возвращает количество элементов, максимально возможное без повторного выделения памяти
c.`reserve(`*num*`)`	Увеличивает емкость, если ее недостаточно[6]
c.`shrink_to_fit()`	Уменьшает емкость до количества элементов (по стандарту C++11)[6]
c1 == *c2*	Возвращает результат проверки того, что вектор *c1* равен вектору *c2* (выполняя операцию == для элементов)
c1 != *c2*	Возвращает результат проверки того, что вектор *c1* не равен вектору *c2* (эквивалент ! (*c1*==*c2*))
c1 < *c2*	Возвращает результат проверки того, что вектор *c1* меньше вектора *c2*

[6] Функции `reserve()` и `shrink_to_fit()` на самом деле в определенном смысле модифицируют векторы, делая некорректными ссылки, указатели и итераторы, установленные на элементы. Однако они перечисляются здесь потому, что не касаются логического содержимого контейнера.

Окончание табл. 7.10

Операция	Действие
c1 > *c2*	Возвращает результат проверки того, что вектор *c1* больше вектора *c2* (эквивалент *c2*<*c1*)
c1 <= *c2*	Возвращает результат проверки того, что вектор *c1* не больше вектора *c2* (эквивалент ! (*c2*<*c1*))
c1 >= *c2*	Возвращает результат проверки того, что вектор *c1* не меньше вектора *c2* (эквивалент ! (*c1*<*c2*))

Присваивание

В табл. 7.11 перечислены способы присваивания новых элементов при удалении старых. Набор функций `assign()` соответствует набору конструкторов. Для присваивания можно использовать разные источники (контейнеры, массивы, стандартный поток ввода), аналогично конструкторам (см. раздел 7.1.2). Все операции присваивания вызывают конструктор по умолчанию, копирующий конструктор, операцию присваивания и/или деструктор типа элементов, в зависимости от того, как изменяется количество элементов.

Таблица 7.11. Операции присваивания векторов

Операция	Действие
c = *c2*	Присваивание всех элементов вектора *c2* вектору *c*
c = *rv*	Перемещающее присваивание всех элементов из вектора *rv* в вектор *c* (по стандарту C++11)
c = *initlist*	Присваивает вектору *c* все элементы списка инициализации *initlist* (по стандарту C++11)
c.assign(*n*,*elem*)	Присваивает *n* копий элемента *elem*
c.assign(*beg*,*end*)	Присваивает элементы из интервала [*beg*,*end*)
c.assign(*initlist*)	Присваивает вектору *c* все элементы из списка инициализации *initlist*
c1.swap(*c2*)	Обменивает элементы векторов *c1* и *c2*
swap(*c1*,*c2*)	Обменивает элементы векторов *c1* и *c2*

Рассмотрим пример:

```
std::list<Elem> l;
std::vector<Elem> coll;
...
// делаем вектор coll копией контейнера l
coll.assign(l.begin(),l.end());
```

Доступ к элементам

Для доступа ко всем элементам вектора следует использовать диапазонный цикл `for` (см. раздел 3.1.4), специальные операции, или итераторы. В табл. 7.12 перечислены все операции над векторами, обеспечивающие прямой доступ к элементам. Как обычно в языках C и C++, первый элемент имеет индекс `0`, а последний — `size()-1`. Таким образом,

n-й элемент имеет индекс n-1. Для неконстантных векторов эти операции возвращают ссылку на элемент. Следовательно, с помощью этих операций можно модифицировать элементы, если это не запрещено по другим причинам.

Таблица 7.12. Прямой доступ к элементам векторов

Операция	Действие
`c[idx]`	Возвращает элемент с индексом `idx` (проверка выхода за пределы допустимого диапазона *не выполняется*)
`c.at(idx)`	Возвращает элемент с индексом `idx` (при выходе за пределы допустимого диапазона *генерирует исключение*)
`c.front()`	Возвращает первый элемент (проверка существования первого элемента *не выполняется*)
`c.back()`	Возвращает последний элемент (проверка существования последнего элемента *не выполняется*)

Самый важный момент для вызывающей стороны состоит в том, выполняют ли эти операции проверку выхода за пределы допустимого диапазона. Эту проверку выполняет только функция `at()`. Если индекс выходит за пределы допустимого диапазона, функция `at()` генерирует исключение `out_of_range` (см. раздел 4.3). Все остальные функции проверку *не производят*. Выход за пределы допустимого диапазона приводит к непредсказуемым последствиям. Выполнение операции `[ ]` и вызов функции `front()` и `back()` для пустого вектора всегда приводят к непредсказуемым последствиям.

```
std::vector<Elem> coll;      // пусто!
coll[5] = elem;              // ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ⇒
                             // непредсказуемые последствия
std::cout << coll.front();   // ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ⇒
                             // непредсказуемые последствия
```

Итак, в сомнительных ситуациях необходимо убедиться, что индекс операции `[ ]` является корректным, или использовать функцию `at()`.

```
std::vector<Elem> coll;      // пусто!

if (coll.size() > 5) {
    coll[5] = elem;          // OK
}

if (!coll.empty()) {
    cout << coll.front();    // OK
}

coll.at(5) = elem;           // генерирует исключение out_of_range
```

Этот код является правильным только в однопоточной среде. В многопоточном контексте необходим механизм синхронизации, предотвращающий модификацию вектора `coll` между проверками его размера и доступом к его элементам (см. раздел 18.4.3).

Функции для итераторов

Векторы предоставляют обычные операции для получения итераторов (табл. 7.13). Итераторы векторов являются итераторами произвольного доступа (см. раздел 9.2). Следовательно, при работе с векторами можно использовать все алгоритмы STL.

Таблица 7.13. Операции над итераторами векторов

Операция	Действие
`c.begin()`	Возвращает итератор произвольного доступа, установленный на первый элемент
`c.end()`	Возвращает итератор произвольного доступа, установленный на позицию, следующую за последним элементом
`c.cbegin()`	Возвращает константный итератор произвольного доступа, установленный на первый элемент (по стандарту C++11)
`c.cend()`	Возвращает константный итератор, установленный на позицию, следующую за последним элементом (по стандарту C++11)
`c.rbegin()`	Возвращает обратный итератор, установленный на первый элемент при обратном обходе
`c.rend()`	Возвращает обратный итератор, установленный на позицию, следующую за последним элементом при обратном обходе
`c.crbegin()`	Возвращает константный обратный итератор, установленный на первый элемент при обратном обходе (по стандарту C++11)
`c.crend()`	Возвращает константный обратный итератор, установленный на позицию, следующую за последним элементом при обратном обходе (по стандарту C++11)

Точный тип этих итераторов определяется реализацией. Однако для векторов итераторы, возвращаемые функциями `begin()`, `cbegin()`, `end()` и `cend()`, часто являются обычными указателями, и это отлично, потому что в классе `vector` скрывается массив в стиле языка C, для которого элементы и обычные указатели образуют интерфейс итераторов произвольного доступа. Однако то, что итераторы представляют собой обычные указатели, использовать невозможно. Например, при работе с безопасной версией библиотеки STL, проверяющей ошибки выхода за пределы допустимого диапазона и другие потенциальные проблемы, тип итератора обычно представляет собой вспомогательный класс. В разделе 9.2.6 продемонстрирована огромная разница между итераторами, реализованными как указатели, и итераторами, реализованными как классы.

Итераторы остаются корректными, пока не произойдет вставка или удаление элемента с меньшим индексом, повторное выделение памяти или изменение емкости (см. раздел 7.3.1).

Вставка и удаление элементов

В табл. 7.14 перечислены операции, предусмотренные для вставки и удаления элементов векторов. Как обычно при использовании библиотеки STL, программист должен гарантировать, что аргументы являются корректными. Итераторы должны ссылаться на корректные позиции, а начало интервала должно предшествовать его концу.

Таблица 7.14. Операции вставки и удаления в векторах

Операция	Действие
c.push_back(*elem*)	Добавляет копию аргумента *elem* в конец вектора
c.pop_back()	Удаляет последний элемент (не возвращая его)
c.insert(*pos*, *elem*)	Вставляет копию аргумента *elem* перед позицией итератора *pos* и возвращает позицию нового элемента
c.insert(*pos*, *n*, *elem*)	Вставляет *n* копий аргумента *elem* перед позицией итератора *pos* и возвращает позицию первого нового элемента (или итератор *pos*, если новых элементов нет)
c.insert(*pos*, *beg*, *end*)	Вставляет копии всех элементов интервала [*beg*,*end*) перед позицией итератора *pos* и возвращает позицию первого нового элемента (или итератор *pos*, если новых элементов нет)
c.insert(*pos*, *initlist*)	Вставляет копии всех элементов списка инициализации *initlist* перед позицией итератора *pos* и возвращает позицию первого нового элемента (или итератор *pos*, если новых элементов нет; по стандарту C++11)
c.emplace(*pos*, *args*...)	Вставляет новый элемент, инициализированный списком аргументов *args* перед позицией итератора *pos*, и возвращает позицию нового элемента (по стандарту C++11)
c.emplace_back(*args*...)	Добавляет в конец вектора новый элемент, инициализированный списком аргументов args (не возвращает ничего; по стандарту C++11)
c.erase(*pos*)	Удаляет элемент в позиции итератора *pos* и возвращает позицию следующего элемента
c.erase(*beg*, *end*)	Удаляет все элементы интервала [*beg*,*end*) и возвращает позицию следующего элемента
c.resize(*num*)	Изменяет количество элементов до *num* (если размер size() увеличивается, новые элементы создаются их конструкторами по умолчанию)
c.resize(*num*, *elem*)	Изменяет количество элементов до *num* (если размер size() увеличивается, новые элементы являются копией аргумента *elem*)
c.clear()	Удаляет все элементы (опустошает контейнер)

Как обычно, программист должен гарантировать, что контейнер не пуст при вызове функции pop_back(). Например:

```
std::vector<Elem> coll; // пусто!

coll.pop_back();        // ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ⇒
                        // непредсказуемые последствия
if (!coll.empty()) {
    coll.pop_back();    // ОК
}
```

Однако следует отметить, что в многопотоковом контексте программист должен гарантировать, что вектор `coll` не изменяется между проверкой его пустоты и выполнением функции `pop_back()` (см. раздел 18.4.3).

С точки зрения производительности программист должен учитывать, что удаление и вставка выполняются быстрее при следующих условиях:

- элементы вставляются и удаляются в конце;
- емкость вектора достаточно велика;
- при одном вызове вставляется сразу несколько элементов, а не по одному при каждом вызове.

Вставка и удаление элементов делает некорректными ссылки, указатели и итераторы, ссылающиеся на элементы, следующие за обрабатываемыми. Вставка, заставляющая выполнить повторное выделение памяти, делает некорректными все ссылки, итераторы и указатели.

Вектор не имеет операции удаления элементов, имеющих конкретное значение. Для выполнения этого действия необходимо использовать алгоритм. Например, следующий код удаляет все элементы, имеющие значение `val`:

```
std::vector<Elem> coll;
...
// удаляем все элементы, имеющие значение val
coll.erase(remove(coll.begin(),coll.end(), val), coll.end());
```

Этот вызов описан в разделе 6.7.1.

Для того чтобы удалить только первый элемент, имеющий конкретное значение, необходимо выполнить следующий код:

```
std::vector<Elem> coll;
...
// удаляем первый элемент со значением val
std::vector<Elem>::iterator pos;
pos = find(coll.begin(),coll.end(), val);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

7.3.3. Использование векторов в качестве массивов языка C

Как и для класса `array<>`, стандартная библиотека C++ гарантирует, что элементы вектора занимают смежные ячейки памяти. Таким образом, можно ожидать, что для любого корректного индекса `i` в векторе `v` следующий код вернет значение `true`:

```
&v[i] == &v[0] + i
```

Из этого вытекает несколько важных последствий. Это значит, что вектор можно использовать во всех ситуациях, в которых применяется динамический массив. Например, вектор можно использовать для хранения данных обычных строк языка C — `char*` или `const char*`.

```
std::vector<char> v;           // создаем вектор как динамический массив символов
v.resize(41);                  // выделяем память для 41 символа (включая '\0')
strcpy(&v[0],"hello, world"); // копируем строку в стиле языка C в вектор
printf("%s\n", &v[0]);         // выводим на экран содержимое вектора
                               // как строку языка C
```

Однако следует отметить, что по стандарту C++11 вы не должны использовать выражение `&a[0]` для получения непосредственного доступа к элементам вектора, потому что для этого предназначена функция-член `data()`.

```
std::vector<char> v;              // создаем динамический массив символов
strcpy(v.data(),"hello, world"); // копируем C-строку в массив
printf("%s\n", v.data());         // выводим на экран содержимое массива
                                  // как C-строку
```

Разумеется, следует быть осторожным при таком использовании вектора (впрочем, как всегда при работе с массивами в стиле языка C и указателями). Например, необходимо убедиться, что размер вектора достаточно большой, чтобы копировать в него данные, а при хранении C-строки в конце вектора записан элемент `'\0'`. Однако этот пример показывает, что если по каким-то причинам вам понадобился массив типа `T`, например, для работы с существующей библиотекой, написанной на языке C, можно использовать класс `vector<T>` и передавать адрес его первого элемента.

Отметим, что нельзя передавать итератор в качестве адреса первого элемента вектора. Тип итераторов векторов зависит от реализации и может совершенно отличаться от указателя.

```
printf("%s\n", v.begin());  // ОШИБКА (может работать, но не переносимо)
printf("%s\n", v.data());   // OK (по стандарту)
printf("%s\n", &v[0]);      // OK, но лучше использовать функцию data()
```

7.3.4. Обработка исключений

Векторы обеспечивают минимальную поддержку для проверки логических ошибок. Единственная функция-член, от которой стандарт требует генерации исключения, — функция `at()`, представляющая собой безопасный вариант операции индексирования (см. раздел 7.3.2). Кроме того, стандарт требует, чтобы генерировались только стандартные исключения, например, `bad_alloc` при недостатке памяти, или исключения, предусмотренные пользовательскими операциями.

Если функции, вызываемые вектором (функции для типа элементов или функции, созданные пользователем), генерируют исключение, то стандартная библиотека C++ предоставляет следующие гарантии.

1. Функции-члены `push_back()` или `emplace_back()` либо выполняются успешно, либо не выполняются совсем. Однако, если перемещающий конструктор не гарантирует отсутствие исключений, то для обеспечения такого поведения следует использовать копирующий, а не перемещающий конструктор. Таким образом, спецификации `nothrow` и `noexcept` для операций, перемещающих элементы, повышают быстродействие программы.
2. Функции-члены `insert()`, `emplace()` и `push_back()` либо выполняются успешно, либо не выполняются совсем, при условии, что операции копирования/

перемещения (конструкторы и операторы присваивания) элементов не генерируют исключений.

3. Функция-член `pop_back()` не генерирует никаких исключений.
4. Функция-член `erase()` не генерирует исключений, если операции копирования/перемещения (конструкторы и операторы присваивания) элементов не генерируют исключений.
5. Функции-члены `swap()` и `clear()` не генерируют исключений.
6. Если используются элементы, никогда не генерирующие исключений при выполнении операций копирования/перемещения (конструкторов и операторов присваивания), то любая операция либо выполняется успешно, либо не выполняется совсем. К таким элементам относятся "простые старые данные" ("plain old data" — POD). Типы данных POD не используют специальных средств языка C++. Например, обычная структура из языка C относится к категории POD.

Все эти гарантии основаны на требовании, что деструкторы не генерируют исключений. Общее обсуждение обработки исключений в библиотеке STL см. в разделе 6.12.2.

7.3.5. Примеры использования векторов

Рассмотрим пример, демонстрирующий простое использование векторов.

```
// cont/vector1.cpp

#include <vector>
#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
   // создаем пустой вектор для строк
   vector<string> sentence;

   // резервируем память для пяти элементов,
   // чтобы избежать повторного выделения памяти
   sentence.reserve(5);

   // добавляем несколько элементов
   sentence.push_back("Hello,");
   sentence.insert(sentence.end(),{"how","are","you","?"});

   // выводим на экран элементы, разделенные пробелами
   copy (sentence.cbegin(), sentence.cend(),
         ostream_iterator<string>(cout," "));
   cout << endl;

   // выводим на экран "технические данные"
   cout << " max_size(): " << sentence.max_size() << endl;
```

```
    cout << " size():     " << sentence.size()     << endl;
    cout << " capacity(): " << sentence.capacity() << endl;

    // меняем местами второй и четвертый элементы
    swap (sentence[1], sentence[3]);

    // вставляем элемент "always" перед элементом "?"
    sentence.insert (find(sentence.begin(),sentence.end(),"?"),
                     "always");

    // присваиваем последнему элементу символ "!"
    sentence.back() = "!";

    // выводим на экран элементы, разделенные пробелами
    copy (sentence.cbegin(), sentence.cend(),
    ostream_iterator<string>(cout," "));
    cout << endl;

    // вновь выводим на экран "технические данные"
    cout << " size()    : " << sentence.size() << endl;
    cout << " capacity(): " << sentence.capacity() << endl;

    // удаляем последние два элемента
    sentence.pop_back();
    sentence.pop_back();

    // уменьшаем емкость (по стандарту C++11)
    sentence.shrink_to_fit();

    // вновь выводим на экран "технические данные"
    cout << " size():   " << sentence.size() << endl;
    cout << " capacity(): " << sentence.capacity() << endl;
}
```

Результат работы программы может выглядеть следующим образом:

```
Hello, how are you ?
max_size(): 1073741823
size():     5
capacity(): 5
Hello, you are how always !
size():     6
capacity(): 10
size():     4
capacity(): 4
```

Обратите внимание на слово "может". Значения функций `max_size()` и `capacity()` не определены и могут изменяться от платформы к платформе. Например, в данном случае мы видим, что реализация вдвое увеличила емкость, обнаружив нехватку памяти, и не всегда уменьшает емкость, несмотря на требование программы.

7.3.6. Класс vector<bool>

Для булевых элементов стандартная библиотека C++ содержит специализацию класса vector<>. Ее предназначение — обеспечить версию вектора, использующую меньший объем памяти по сравнению с обычной реализацией класса vector<> для типа bool. Обычная реализация резервирует по крайней мере один байт для каждого элемента. Специализация vector<bool>, как правило, использует только один бит для хранения каждого элемента, т.е. в восемь раз меньше. Однако у такой реализации тоже есть недостаток: в языке C++ наименьшей адресуемой единицей памяти является один байт. Таким образом, эта специализация вектора требует особого способа работы со ссылками и итераторами.

В результате класс vector<bool> не соответствует всем требованиям, предъявляемым к другим векторам. Например, значение vector<bool>::reference не может стоять в левой части операции присваивания, а итератор vector<bool>::iterator не является итератором произвольного доступа. Следовательно, шаблонный код может работать с векторами любого типа, кроме bool. Кроме того, производительность класса vector<bool> может уступать обычным реализациям, поскольку операции над элементами должны быть преобразованы в операции над битами. В то же время реализация vector<bool> зависит от конкретного компилятора. Таким образом, производительность (т.е. быстродействие и использование памяти) может варьироваться.

Отметим, что класс vector<bool> представляет собой нечто бо́льшее, чем специализацию класса vector<> для типа bool. Он также содержит специальные операции для битов. Это позволяет более удобно работать с битами и флагами.

Объект класса vector<bool> имеет динамический размер, поэтому его можно рассматривать как битовое поле с динамическим размером. Таким образом, можно добавлять и удалять биты. Если вам необходимо битовое поле со статическим размером, следует использовать класс bitset, а не vector<bool>. Класс bitset описан в разделе 12.5.

Дополнительные операции в классе vector<bool> перечислены в табл. 7.15.

Таблица 7.15. Специальные операции в классе vector<bool>

Операция	Действие
`c.flip()`	Отрицание всех булевых элементов (т.е. дополнение ко всем битам)
`c[`*idx*`].flip()`	Отрицание булевого элемента с индексом *idx* (дополнение отдельного бита)
`c[`*idx*`]` = *val*	Присваивание значения *val* булевому элементу с индексом *idx* (присваивание отдельного бита)
`c[`*idx1*`]` = `c[`*idx2*`]`	Присваивание значения элемента с индексом *idx2* элементу с индексом *idx1*

Операция flip(), вычисляющая дополнение, может вызываться как для всех битов, так и для отдельного бита в векторе. Последний пункт вызывает интерес, потому что можно ожидать, что операция [ ] вернет объект типа bool и что вызов функции flip() для такого основного типа не предусмотрен. Здесь класс vector<bool> использует универсальный трюк под названием *прокси*[7]. Для класса vector<bool> тип значения, возвращаемого операцией индексирования (и другими операциями, возвращающими элемент), является вспо-

[7] Прокси позволяет управлять тем, чем обычно управлять невозможно. Он часто используется для обеспечения большей безопасности. В данном случае прокси обеспечивает управление, позволяющее выполнять определенные операции, хотя в принципе возвращаемое значение ведет себя как объект типа bool.

могательным. Если необходимо, чтобы возвращаемое значение имело тип bool, используется автоматическое преобразование. Для других операций предусмотрены функции-члены. Соответствующая часть объявления класса vector<bool> приведена ниже.

```
namespace std {
   template <typename Allocator> class vector<bool,Allocator> {
      public:
         // вспомогательный прокси-тип для модификация элементов:
         class reference {
         ...
         public:
            reference& operator= (const bool) noexcept; // присваивание
            reference& operator= (const reference&) noexcept;
            operator bool() const noexcept; // автоматическое преобразование
                                            // типа в bool
            void flip() noexcept;           // дополнение бита
      };
     ...

      // операции для доступа к элементам возвращают прокси-тип
      // reference, а не тип bool:
      reference operator[](size_type idx);
      reference at(size_type idx);
      reference front();
      reference back();
      ...
   };
}
```

Легко видеть, что все функции-члены для доступа к элементу возвращают тип reference. Следовательно, можно писать следующий код:

```
c.front().flip(); // отрицание первого булева элемента
c[5] = c.back();  // присваиваем последний элемент элементу с индексом 5
```

Как обычно, чтобы избежать непредсказуемых последствий, вызывающая сторона должна гарантировать, что первый, шестой и последний элементы существуют.

Отметим, что внутренний тип прокси reference используется только в неконстантных контейнерах типа vector<bool>. Константные функции-члены для доступа к элементам возвращают значение типа const_reference, который является определением типа bool.

7.4. Деки

Дек очень похож на вектор. Он управляет своими элементами с помощью динамического массива, обеспечивает прямой доступ и имеет практически такой же интерфейс, как и вектор. Разница заключается в том, что при работе с деком динамический массив открыт с обеих сторон. Следовательно, дек быстро выполняет вставки и удаления как в конце, так и в начале (рис. 7.3).

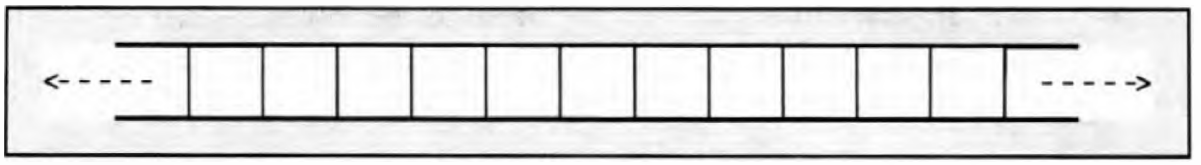


Рис. 7.3. Логическая структура дека

Для того чтобы обеспечить эту возможность, дек обычно реализуется как пакет отдельных блоков, в котором первый блок растет в одном направлении, а последний — в противоположном (рис. 7.4).

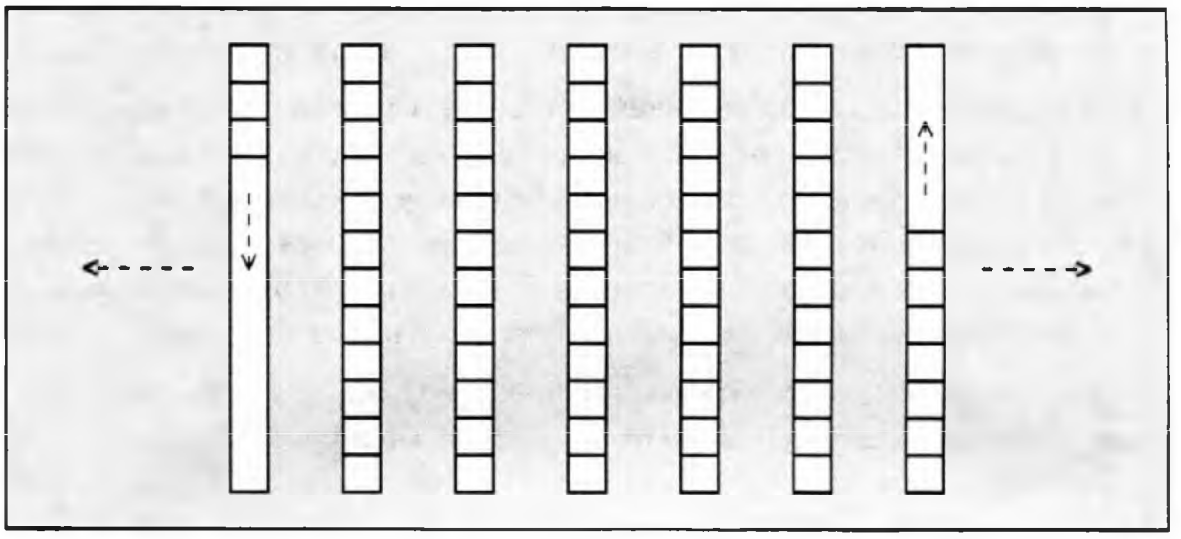


Рис. 7.4. Внутренняя структура дека

Для использования дека необходимо включить в программу заголовочный файл `<deque>`:

```
#include <deque>
```

где тип определен как шаблонный класс в пространстве имен `std`.

```
namespace std {
    template <typename T,
              typename Allocator = allocator<T> >
    class deque;
}
```

Как и во всех последовательных контейнерах, тип элементов передается как первый шаблонный параметр. Необязательный второй параметр зависит от модели памяти, по умолчанию он представляет собой `allocator` (см. главу 19).

7.4.1. Возможности деков

Возможности деков отличаются от возможностей векторов.

- Вставка и удаление элементов выполняются быстро как в начале, так и в конце дека (в векторе эти операции быстро выполняются только в конце). Эти операции выполняются за амортизированное константное время.

- Внутренняя структура имеет еще один уровень косвенности для доступа к элементам, поэтому в деках доступ к элементам и перемещение итератора обычно выполняется немного медленнее.
- Итераторы должны быть интеллектуальными указателями специального типа, а не обычными указателями, потому что они должны перемещаться между разными блоками.
- В системах, ограничивающих размеры блоков памяти (например, в некоторых операционных системах персональных компьютеров), дек может содержать больше элементов, потому что он использует больше одного блока памяти. Следовательно, значение, возвращаемое функцией-членом `max_size()`, у деков может быть больше.
- Деки не поддерживают управление емкостью и не определяют момент повторного выделения памяти. В частности, любая вставка или удаление элементов, выполняемые не в начале или не в конце дека, делает некорректными все указатели, ссылки и итераторы, ссылающиеся на элементы дека. Однако повторное выделение памяти может выполняться быстрее, чем у векторов, поскольку благодаря своей внутренней структуре деки не обязаны копировать все элементы при повторном выделении памяти.
- Блоки памяти можно освобождать, если они больше не нужны, поэтому размер дека может уменьшаться (однако произойдет ли это и как это будет осуществлено, зависит от реализации).

Деки имеют следующие общие свойства с векторами.

- Вставка и удаление элементов в середине дека выполняется относительно медленно, поскольку все элементы вплоть до конца должны быть перемещены, чтобы освободить место или заполнить просвет.
- Итераторы дека являются итераторами произвольного доступа.

Итак, дек следует выбирать в следующих ситуациях.

- Если вы вставляете или удаляете элементы с обоих концов (классический случай — очередь).
- Вы не собираетесь ссылаться на элементы контейнера.
- Важно, чтобы контейнер освобождал память, которая больше не используется (однако стандарт не гарантирует, что это произойдет).

Интерфейсы векторов и деков практически совпадают, поэтому их использование не вызывает проблем, если от вектора и дека не требуется ничего особенного.

7.4.2. Операции над деком

В табл. 7.16–7.18 перечислены все операции, предусмотренные для деков[8].

[8] Функция `shrink_to_fit()` по сути модифицирует дек, потому что делает некорректными ссылки, указатели и итераторы на элементы. Однако она указана как немодифицирующая, потому что он не затрагивает логическое содержимое контейнера.

Таблица 7.16. Конструкторы и деструкторы деков

Операция	Действие
`deque<Elem> c`	Конструктор по умолчанию; создает пустой дек, не содержащий никаких элементов
`deque<Elem> c(c2)`	Копирующий конструктор; создает новый дек как копию объекта *c2* (все элементы копируются)
`deque<Elem> c = c2`	Копирующий конструктор; создает новый дек как копию объекта *c2* (все элементы копируются)
`deque<Elem> c(rv)`	Перемещающий конструктор; создает новый дек, получая содержимое объекта *rv* (по стандарту C++11)
`deque<Elem> c = rv`	Перемещающий конструктор; создает новый дек, получая содержимое объекта *rv* (по стандарту C++11)
`deque<Elem> c(n)`	Создает дек, содержащий *n* элементов, созданных конструктором по умолчанию
`deque<Elem> c(n, elem)`	Создает дек, инициализированный *n* копиями элемента *elem*
`deque<Elem> c(beg, end)`	Создает дек, инициализированный элементами интервала [*beg*,*end*)
`deque<Elem> c(initlist)`	Создает дек, инициализированный элементами списка инициализации *initlist* (по стандарту C++11)
`deque<Elem> c = initlist`	Создает дек, инициализированный элементами списка инициализации *initlist* (по стандарту C++11)
`c.~deque()`	Уничтожает все элементы и освобождает память

Операции над деками отличаются от операций над векторами только двумя особенностями.

1. Деки не имеют функций для емкости (`capacity()` и `reserve()`).
2. Деки имеют специальные функции для вставки и удаления первого элемента (`push_front()` и `pop_front()`).

Поскольку остальные операции такие же, как в векторе, мы не будем их объяснять повторно. Их описание приводится в разделе 7.3.2.

Отметим, однако, что в стандарт C++11 была добавлена функция-член `shrink_to_fit()`, представляющая собой несвязывающий запрос на уменьшение внутренней памяти, чтобы она соответствовала количеству элементов. Читатели могут возразить, что функция-член `shrink_to_fit()` для деков не имеет смысла, поскольку им позволено освобождать блоки памяти. Тем не менее память, содержащая все указатели на блоки памяти, обычно не уменьшается. Эту ситуацию можно исправить с помощью вызова функции-члена `shrink_to_fit()`.

Таблица 7.17. Немодифицирующие операции над деками

Операция	Действие
`c.empty()`	Возвращает результат проверки, что контейнер пуст (эквивалент `size()==0`, но может работать быстрее)
`c.size()`	Возвращает текущее количество элементов

Окончание табл. 7.17

Операция	Действие
`c.max_size()`	Возвращает максимально возможное количество элементов
`c.shrink_to_fit()`	Запрос на уменьшение емкости, чтобы емкость контейнера соответствовала количеству элементов (начиная со стандарта C++11)[8]
c1 == *c2*	Возвращает результат проверки, что дек *c1* равен деку *c2* (выполняет операцию == к элементам)
c1 != *c2*	Возвращает результат проверки, что дек *c1* не равен деку *c2* (эквивалент ! (*c1*==*c2*))
c1 < *c2*	Возвращает результат проверки, что дек *c1* меньше дека *c2*
c1 > *c2*	Возвращает результат проверки, что дек *c1* больше дека *c2* (эквивалент *c2*<*c1*)
c1 <= *c2*	Возвращает результат проверки, что дек *c1* не больше дека *c2* (эквивалент ! (*c2*<*c1*))
c1 >= *c2*	Возвращает результат проверки, что дек *c1* не меньше дека *c2* (эквивалент ! (*c1*<*c2*))
`c[`*idx*`]`	Возвращает элемент с индексом *idx* (*без* проверки интервала)
`c.at(`*idx*`)`	Возвращает элемент с индексом *idx* (генерирует исключение, если *idx* выходит за пределы интервала)
`c.front()`	Возвращает первый элемент (*не* проверяет, существует ли этот элемент)
`c.back()`	Возвращает последний элемент (*не* проверяет, существует ли этот элемент)
`c.begin()`	Возвращает итератор произвольного доступа, установленный на первый элемент
`c.end()`	Возвращает итератор произвольного доступа, установленный на позицию, следующую за последним элементом
`c.cbegin()`	Возвращает константный итератор произвольного доступа, установленный на первый элемент (по стандарту C++11)
`c.cend()`	Возвращает константный итератор произвольного доступа, установленный на позицию, следующую за последним элементом (по стандарту C++11)
`c.rbegin()`	Возвращает обратный итератор, установленный на первый элемент при обратном обходе
`c.rend()`	Возвращает обратный итератор, установленный на позицию, следующую за последним элементом при обратном обходе
`c.crbegin()`	Возвращает константный обратный итератор, установленный на первый элемент при обратном обходе (по стандарту C++11)
`c.crend()`	Возвращает константный обратный итератор, установленный на позицию, следующую за последним элементом при обратном обходе (по стандарту C++11)

Кроме того, следует учитывать следующие факты.

1. Ни одна функция-член, обеспечивающая доступ к элементам (за исключением функции-члена `at()`), не проверяет, является ли индекс или итератор корректным.
2. Вставка или удаление может вызывать повторное выделение памяти. Таким образом, любая вставка или удаление делает некорректными все указатели, ссылки и итераторы, установленные на другие элементы дека. Исключением является ситуация, когда элемент вставляется в начало или в конец. В этом случае ссылки и указатели остаются корректными, но итераторы — нет.

Таблица 7.18. Модифицирующие операции над деками

Операция	Действие
c = *c2*	Присваивает все элементы дека *c2* деку *c*
c = *rv*	Перемещающее присваивание всех элементов дека *rv* деку *c* (по стандарту C++11)
c = *initlist*	Присваивает все элементы списка инициализации *initlist* деку *c* (по стандарту C++11)
c.assign(*n*, *elem*)	Присваивает *n* копий элемента *elem*
c.assign(*beg*, *end*)	Присваивает элементы интервала [*beg*,*end*)
c.assign(*initlist*)	Присваивает все элементы списка инициализации *initlist*
c1.swap(*c2*)	Обменивает данные деков *c1* и *c2*
swap(*c1*, *c2*)	Обменивает данные деков *c1* и *c2*
c.push_back(*elem*)	Добавляет копию элемента *elem* в конец дека
c.pop_back()	Удаляет последний элемент (не возвращая его)
c.push_front(*elem*)	Вставляет копию элемента *elem* в начало дека
c.pop_front()	Удаляет первый элемент (не возвращая его)
c.insert(*pos*, *elem*)	Вставляет копию элемента *elem* перед позицией итератора *pos* и возвращает позицию нового элемента
c.insert(*pos*, *n*, *elem*)	Вставляет *n* копий элементов *elem* перед позицией итератора *pos* и возвращает позицию нового элемента (или итератор *pos*, если нового элемента нет)
c.insert(*pos*, *beg*, *end*)	Вставляет копию всех элементов интервала [*beg*,*end*) перед позицией итератора *pos* и возвращает позицию первого нового элемента (или позицию *pos*, если нового элемента нет)
c.insert(*pos*, *initlist*)	Вставляет копию всех элементов списка инициализации *initlist* перед позицией итератора *pos* и возвращает позицию первого нового элемента (или позицию *pos*, если нового элемента нет; по стандарту C++11)
c.emplace(*pos*, *args*...)	Вставляет новый элемент, инициализированный списком аргументов *args* перед позицией итератора *pos* и возвращает позицию нового элемента (по стандарту C++11)

Окончание табл. 7.18

Операция	Действие
`c.emplace_back(args...)`	Добавляет новый элемент, инициализированный списком аргументов *args*, в конец дека (не возвращает ничего; по стандарту C++11)
`c.emplace_front(args...)`	Вставляет новый элемент элемента, инициализированный списком аргументов *args*, в начало дека (не возвращает ничего; по стандарту C++11)
`c.erase(pos)`	Удаляет элемент, стоящий в позиции итератора *pos*, и возвращает позицию следующего элемента
`c.erase(beg,end)`	Удаляет все элементы интервала [*beg*,*end*) и возвращает позицию следующего элемента
`c.resize(num)`	Изменяет количество элементов до *num* (если размер `size()` увеличивается, новые элементы создаются их конструкторами по умолчанию)
`c.resize(num,elem)`	Изменяет количество элементов на *num* (если размер `size()` увеличивается, новые элементы являются копиями элемента *elem*)
`c.clear()`	Удаляет все элементы (опустошает контейнер)

7.4.3. Обработка исключений

В принципе деки обрабатывают исключения точно так же, как векторы (см. раздел 7.3.4). Дополнительные операции `push_front()`, `emplace_front()` и `pop_front()` соответствуют функциям-членам `push_back()`, emplace_back() и `pop_back()` соответственно. Таким образом, стандартная библиотека C++ обеспечивает следующее.

- Если элемент вставлен с помощью функций-членов push_front(), emplace_front(), push_back() или emplace_back() и возникло исключение, эти функции не выполняют никаких действий. И вновь, перемещающие операции, гарантирующие отсутствие исключений, повышают быстродействие.
- Ни `pop_back()`, ни `pop_front()` не генерируют никаких исключений.

Общее обсуждение обработки исключений в библиотеке STL приведено в разделе 6.12.2.

7.4.4. Примеры использования деков

Возможности деков демонстрирует программа, приведенная ниже.

```
// cont/deque1.cpp

#include <iostream>
#include <deque>
#include <string>
#include <algorithm>
#include <iterator>
```

```
using namespace std;

int main()
{
   // создаем пустой дек, содержащий строки
   deque<string> coll;

   // вставляем несколько элементов
   coll.assign (3, string("string"));
   coll.push_back ("last string");
   coll.push_front ("first string");

   // выводим на экран элементы, разделенные символами перехода на новую строку
   copy (coll.cbegin(), coll.cend(),
         ostream_iterator<string>(cout,"\n"));
   cout << endl;

   // удаляем первый и последний элементы
   coll.pop_front();
   coll.pop_back();

   // вставляем слово "another" перед каждым элементом, кроме первого
   for (unsigned i=1; i<coll.size(); ++i) {
        coll[i] = "another " + coll[i];
   }

   // изменяем размер на четыре элемента
   coll.resize (4, "resized string");

   // выводим на экран элементы, разделенные символами перехода на новую строку
   copy (coll.cbegin(), coll.cend(),
         ostream_iterator<string>(cout,"\n"));
}
```

Результаты работы программы выглядят следующим образом:

```
first string
string
string
string
last string

string
another string
another string
resized string
```

7.5. Списки

Список (экземпляр контейнерного класса `list<>`) управляет своими элементами как двусвязный список (рис. 7.5). Как обычно, стандартная библиотека C++ не указывает вид реализации, но это следует из названия списка, его ограничений и спецификаций.

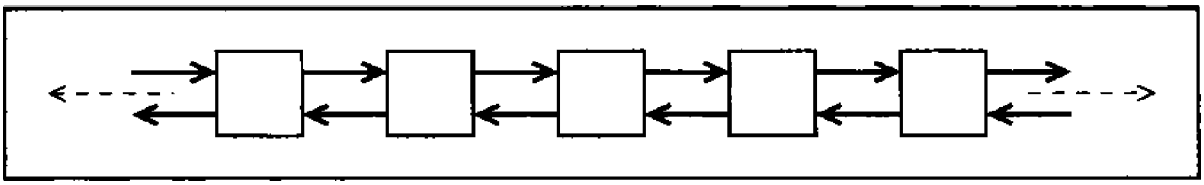


Рис. 7.5. Структура списка

Для использования списка в программу необходимо включить заголовочный файл `<list>`:

```
#include <list>
```

В этом файле тип определен как шаблонный класс в пространстве `std`.

```
namespace std {
   template <typename T,
             typename Allocator = allocator<T> >
   class list;
}
```

Элементы списка могут иметь любой тип `T`. Необязательный второй шаблонный параметр определяет модель памяти (см. главу 19). По умолчанию используется модель памяти `allocator`, поддерживаемая стандартной библиотекой C++.

7.5.1. Возможности списков

Внутренняя структура списка совершенно отлична от внутренней структуры класса `array`, вектора или дека. Объект списка содержит два указателя, так называемых *якоря*, ссылающихся на первый и последний элементы. Каждый элемент содержит указатель на предыдущий и следующий элементы (или на якорь). Для вставки нового элемента достаточно выполнить операции над соответствующими указателями (рис. 7.6).

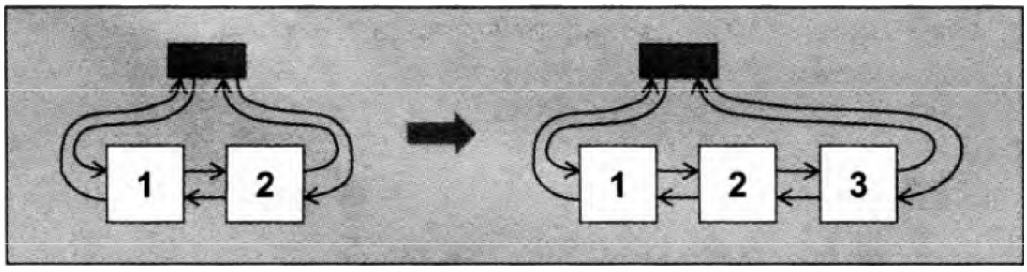


Рис. 7.6. Внутренняя структура списка при добавлении значения

Таким образом, список отличается от массивов, векторов и деков следующими аспектами.

- Список не предоставляет произвольный доступ. Например, для доступа к пятому элементу необходимо пройти через первые четыре элемента, следуя по цепочке связей. Таким образом, доступ к произвольному элементу списка выполняется медленно. Однако список можно обходить с обоих концов. Следовательно, доступ к первому и последнему элементам осуществляется быстро.

- Вставка и удаление элементов выполняется быстро в любой позиции (при условии, что итератор уже установлен на эту позицию), а не только с концов. Вставка и удаление элементов всегда выполняется за константное время, потому что никакие другие элементы перемещать не требуется. Все операции сводятся к манипуляциям указателями.
- Вставка и удаление элементов не делает некорректными указатели, ссылки и итераторы, установленные на другие элементы.
- Список поддерживает обработку исключений таким образом, что почти каждая операция либо выполняется успешно, либо не выполняется вообще. Таким образом, список не может оказаться в промежуточном положении из-за незавершенной операции.

Функции-члены списка отражают его отличия от массивов, векторов и деков.

- Списки содержат функции-члены `front()`, `push_front()` и `pop_front()`, а также `back()`, `push_back()` и `pop_back()`.
- Списки не имеют оператора индексирования и функции-члена `at()`, потому что они не поддерживают произвольный доступ.
- Списки не содержат операции для изменения емкости или перераспределения памяти, потому что они не нужны. Каждый элемент занимает свою собственную ячейку памяти, которая остается корректной, пока элемент не будет удален.
- Списки содержат много специальных функций-членов для перемещения и удаления элементов. Эти функции-члены представляют собой более быстрые версии общих алгоритмов, имеющих те же имена. Они работают быстрее, потому что они лишь перенаправляют указатели, а не копируют и не перемещают значения.

7.5.2. Операции над списками

Создание, копирование и удаление

Возможность создавать, копировать и уничтожать списки ничем не отличается от возможностей других последовательных контейнеров. Список этих операций над списком приведен в табл. 7.19. Некоторые замечания о возможных источниках инициализации приведены в разделе 7.1.2.

Таблица 7.19. Конструкторы и деструкторы списков

Операция	Действие
`list<Elem>` *c*	Конструктор по умолчанию; создает пустой список, не содержащий никаких элементов
`list<Elem>` *c*(*c2*)	Копирующий конструктор; создает новый список как копию списка *c2* (копируются все элементы)

Окончание табл. 7.19

Операция	Действие
`list<Elem>` *c = c2*	Копирующий конструктор; создает новый список как копию списка *c2* (копируются все элементы)
`list<Elem>` *c (rv)*	Перемещающий конструктор; создает новый список, получающий содержимое списка *rv* (по стандарту C++11)
`list<Elem>` *c = rv*	Перемещающий конструктор; создает новый список, получающий содержимое списка *rv* (по стандарту C++11)
`list<Elem>` *c (n)*	Создает список из *n* элементов, созданных конструктором их типа по умолчанию
`list<Elem>` *c (n, elem)*	Создает список, инициализированный *n* копиями элемента *elem*
`list<Elem>` *c (beg, end)*	Создает список, инициализированный элементами интервала [*beg,end*)
`list<Elem>` *c (initlist)*	Создает список, инициализированный элементами списка инициализации *initlist* (по стандарту C++11)
`list<Elem>` *c = initlist*	Создает список, инициализированный элементами списка инициализации *initlist* (по стандарту C++11)
c.`~list()`	Уничтожает все элементы списка и освобождает память

Немодифицирующие операции

Списки предусматривают обычные операции для работы с размером и для сравнения. Эти операции перечислены в табл. 7.20 и обсуждаются в разделе 7.1.2.

Таблица 7.20. Немодифицирующие операции над списками

Операция	Действие
c.`empty()`	Возвращает результат проверки того, что контейнер пуст (эквивалент `size()==0`, но может работать быстрее)
c.`size()`	Возвращает текущее количество элементов
c.`max_size()`	Возвращает максимально возможное количество элементов
c1 == c2	Возвращает результат проверки, что список *c1* равен списку *c2* (выполняет оператор == для элементов)
c1 != c2	Возвращает результат проверки, что список *c1* не равен списку *c2* (эквивалент `!` (*c1==c2*))
c1 < c2	Возвращает результат проверки, что список *c1* меньше списка *c2*.
c1 > c2	Возвращает результат проверки, что список *c1* больше списка *c2* (эквивалент *c2<c1*)
c1 <= c2	Возвращает результат проверки, что список *c1* не больше списка *c2* (эквивалент `!` (*c2<c1*))
c1 >= c2	Возвращает результат проверки, что список *c1* не меньше списка *c2* (эквивалент `!` (*c1<c2*))

Присваивания

Списки также предусматривают операции присваивания, типичные для последовательных контейнеров (табл. 7.21). Как обычно, операции вставки соответствуют конструкторам для обеспечения разных источников инициализации (см. раздел 7.1.2).

Таблица 7.21. Операции присваивания для списков

Операция	Действие
c = c2	Присваивание всех элементов списка *c2* списку *c*
c = rv	Перемещение всех элементов списка *rv* в список *c* (по стандарту C++11)
c = initlist	Присваивание всех элементов списка инициализации *initlist* списку *c* (по стандарту C++11)
`c.assign(n,elem)`	Присваивание *n* копий элемента *elem* списку *c*
`c.assign(beg,end)`	Присваивание списку *c* элементов интервала [*beg*,*end*)
`c.assign(initlist)`	Присваивание всех элементов списка инициализации *initlist* списку *c*
`c1.swap(c2)`	Обмен данных между списками *c1* и *c2*
`swap(c1,c2)`	Обмен данных между списками *c1* и *c2*

Доступ к элементам

Для доступа ко всем элементам списка необходимо использовать диапазонные циклы `for` (см. раздел 3.1.4), специальные операции или итераторы. Поскольку списки не поддерживают произвольный доступ к элементам в принципе, в них предусмотрены две функции-члены — `front()` и `back()`, — предоставляющие непосредственный доступ только к первому и последнему элементам (табл. 7.22).

Таблица 7.22. Прямой доступ к элементам списка

Операция	Действие
`c.front()`	Возвращает первый элемент (*не* проверяя, существует ли этот элемент)
`c.back()`	Возвращает последний элемент (*не* проверяя, существует ли этот элемент)

Как обычно, эти операции *не* проверяют, пуст ли контейнер. Если контейнер пуст, вызов этих операций приводит к непредсказуемым последствиям. Таким образом, вызывающая сторона должна гарантировать, что контейнер содержит хотя бы один элемент. Например:

```
std::list<Elem> coll;          // пусто!

std::cout << coll.front();     // ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ⇒
                               // непредсказуемые последствия
if (!coll.empty()) {
    std::cout << coll.back(); // OK
}
```

Отметим, что этот код корректно работает только в однопоточной среде. В многопоточной среде необходимы механизмы синхронизации, гарантирующие, что список `coll` не будет изменен между проверкой его размера и доступом к элементу (см. раздел 18.4.3).

Функции для итераторов

Для доступа ко всем элементам списка необходимо использовать итераторы. Списки содержат обычные функции (табл. 7.23). Однако, поскольку список не обеспечивает произвольного доступа к элементам, эти итераторы могут быть лишь двунаправленными.

Таким образом, к спискам нельзя применять алгоритмы, требующие итераторов произвольного доступа. К этой категории относятся все алгоритмы, интенсивно оперирующие порядком элементов, особенно алгоритмы сортировки. Однако для сортировки элементов списки содержат специальную функцию-член `sort()` (см. раздел 8.8.1).

Таблица 7.23. Функции для работы с итераторами списка

Операция	Действие
`c.begin()`	Возвращает двунаправленный итератор, установленный на первый элемент
`c.end()`	Возвращает двунаправленный итератор, установленный на позицию, следующую за последним элементом
`c.cbegin()`	Возвращает константный двунаправленный итератор, установленный на первый элемент (по стандарту C++11)
`c.cend()`	Возвращает константный двунаправленный итератор, установленный на позицию, следующую за последним элементом (начиная со стандарта C++11)
`c.rbegin()`	Возвращает обратный итератор, установленный на первый элемент при обратном обходе
`c.rend()`	Возвращает обратный итератор, установленный на позицию, следующую за последним элементом при обратном обходе
`c.crbegin()`	Возвращает константный обратный итератор, установленный на первый элемент при обратном обходе (по стандарту C++11)
`c.crend()`	Возвращает константный обратный итератор, установленный на позицию, следующую за последним элементом при обратном обходе (по стандарту C++11)

Вставка и удаление элементов

В табл. 7.24 перечислены операции для вставки и удаления элементов списка. Список содержит все функции, предусмотренные для дека, дополненные специальными реализациями алгоритмов `remove()` и `remove_if()`.

Таблица 7.24. Операции вставки и удаления элементов списка

Операция	Действие
`c.push_back(`*elem*`)`	Добавляет копию аргумента *elem* в конец списка
`c.pop_back()`	Удаляет последний элемент (не возвращая его)

Окончание табл. 7.24

Операция	Действие
c.push_front(*elem*)	Вставляет копию аргумента *elem* в начало списка
c.pop_front()	Удаляет первый элемент (не возвращая его)
c.insert(*pos*, *elem*)	Вставляет копию аргумента *elem* перед позицией итератора *pos* и возвращает позицию нового элемента
c.insert(*pos*, *n*, *elem*)	Вставляет *n* копий аргумента *elem* перед позицией итератора *pos* и возвращает позицию первого нового элемента (или *pos*, если нового элемента нет)
c.insert(*pos*, *beg*, *end*)	Вставляет копии всех элементов интервала [*beg*,*end*) перед позицией итератора *pos* и возвращает первый новый элемент (или позицию *pos*, если нового элемента нет)
c.insert(*pos*, *initlist*)	Вставляет копии всех элементов списка инициализации *initlist* перед позицией итератора *pos* и возвращает позицию первого элемента (или позицию *pos*, если нового элемента нет; по стандарту C++11)
c.emplace(*pos*, *args*...)	Вставляет новый элемент, инициализированный списком аргументов args перед позицией итератора *pos* и возвращает позицию нового элемента (по стандарту C++11)
c.emplace_back(*args*...)	Добавляет в конец списка новый элемент, инициализированный списком аргументов args (ничего не возвращая; по стандарту C++11)
c.emplace_front(*args*...)	Вставляет в начало списка новый элемент, инициализированный списком аргументов args (ничего не возвращая; по стандарту C++11)
c.erase(*pos*)	Удаляет элемент, занимающий позицию итератора *pos*, и возвращает позицию следующего элемента
c.erase(*beg*, *end*)	Удаляет все элементы интервала [*beg*,*end*) и возвращает позицию следующего элемента
c.remove(val)	Удаляет все элементы, имеющие значение val
c.remove_if(*op*)	Удаляет все элементы, для которых операция *op*(elem) возвращает true
c.resize(*num*)	Изменяет количество элементов до *num* (если размер size() увеличивается, новые элементы создаются их конструкторами по умолчанию)
c.resize(*num*, *elem*)	Изменяет количество элементов до *num* (если размер size() увеличивается, новые элементы представляют собой копии объекта *elem*)
c.clear()	Удаляет все элементы (опустошает контейнер)

Как обычно при использовании библиотеки STL, программист должен гарантировать, что аргументы являются корректными. Итераторы должны ссылаться на корректные позиции, а начало интервала должно предшествовать его концу.

Вставка и удаление выполняются быстрее, если при работе с несколькими элементами мы используем один вызов операции для нескольких элементов, а не несколько вызовов.

Для удаления элементов списки предусматривают специальные реализации алгоритмов `remove()` (см. раздел 11.7.1). Эти функции-члены работают быстрее, чем алгоритмы `remove()`, потому что они манипулируют только внутренними указателями, а не элементами. Итак, в отличие от векторов и деков, при работе со списками следует использовать функцию-член `remove()`, а не сам алгоритм (см. раздел 7.3.2). Для удаления всех элементов, имеющих конкретное значение, можно сделать следующее (см. раздел 6.7.3):

```
std::list<Elem> coll;
...
// удаляем все элементы, имеющие значение val
coll.remove(val);
```

Однако, для того чтобы удалить только первое вхождение этого значения, необходимо применить алгоритм, такой как описанный в разделе 7.3.2, посвященном векторам.

Можно определить критерий удаления элементов с помощью функции или функционального объекта и использовать его в функции-члене `remove_if()`. Функция `remove_if()` удаляет каждый элемент, для которого передаваемая операция возвращает значение `true`. Примером использования функции-члена `remove_if()` является код, удаляющий все элементы, имеющие четные значения.

```
// удаление элементов, имеющих четное значение
coll.remove_if ([] (int i) {
                    return i % 2 == 0;
                });
```

Здесь для поиска удаляемых элементов используется лямбда-функция. Поскольку лямбда-функция возвращает `true`, когда передаваемый элемент имеет четное значение, приведенный код удаляет все четные элементы. Дополнительные примеры использования функций-членов `remove()` и `remove_if()` описаны в разделе 11.7.1.

Следующие операции не делают некорректными итераторы и ссылки на другие элементы: `insert()`, `emplace()`, `emplace...()`, `push_front()`, `push_back()`, `pop_front()`, `pop_back()` и `erase()`.

Функции срезки и функции, изменяющие порядок элементов

Связанные списки имеют одно преимущество: удаление и вставка элементов в любой позиции выполняются за константное время. При переносе элементов из одного контейнера в другой это преимущество усиливается, потому что для этого достаточно перенаправить внутренние указатели (рис. 7.7).

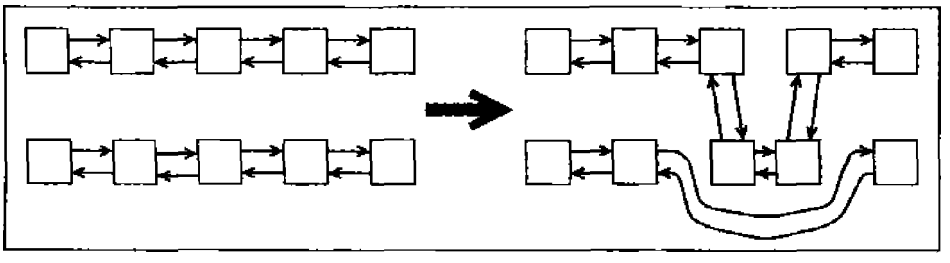


Рис. 7.7. Операции срезки, изменяющие порядок элементов списка

Для поддержки этой возможности списки содержат не только функцию-член `remove()`, но и дополнительные модифицирующие функции-члены для изменения порядка и изменения связей между элементами и интервалами. Эти операции можно выполнять для перемещения элементов в списке или между двумя списками, при условии, что списки имеют одинаковые типы.

Эти функции перечислены в табл. 7.25 и подробно описаны в разделе 8.8.

Таблица 7.25. Специальные модифицирующие операции над списками

Операция	Действие
`c.unique()`	Удаляет дубликаты последовательных элементов, имеющих одинаковые значения
`c.unique(`*op*`)`	Удаляет дубликаты последовательных элементов, для которых операция *op*`()` возвращает `true`
`c.splice(`*pos*`,`*c2*`)`	Перемещает все элементы списка *c2* в список *c* и размещает их перед позицией итератора *pos*
`c.splice(`*pos*`,`*c2*`,`*c2pos*`)`	Перемещает элемент, занимающий позицию *c2pos* в списке *c2*, в список *c* и размещает его перед позицией *pos* (списки *c* и *c2* могут совпадать)
`c.splice(`*pos*`,`*c2*`,`*c2beg*`,`*c2end*`)`	Перемещает все элементы интервала [*c2beg*,*c2end*) списка *c2* в список *c* и размещает их перед позицией *pos* списка *c* (списки *c* и *c2* могут совпадать)
`c.sort()`	Упорядочивает все элементы с помощью оператора <
`c.sort(`*op*`)`	Упорядочивает все элементы с помощью функционального объекта *op*`()`
`c.merge(`*c2*`)`	Если оба контейнера содержат упорядоченные элементы, перемещает все элементы списка *c2* в список *c*, так что все элементы сливаются и остаются упорядоченными
`c.merge(`*c2*`,`*op*`)`	Если оба контейнера содержат упорядоченные элементы в соответствии с критерием *op*(), перемещает все элементы списка *c2* в список *c*, так что все элементы сливаются и остаются упорядоченными в соответствии с критерием *op*()
`c.reverse()`	Изменяет порядок следования всех элементов на противоположный

7.5.3. Обработка исключений

Списки обеспечивают наилучшую поддержку безопасности исключений по сравнению с остальными стандартными контейнерами STL. Почти все операции над списками либо успешны, либо не имеют последствий. Единственными операциями, не дающими гарантий относительно исключений, являются операции присваивания и функция-член `sort()` (они дают обычные "базовые гарантии", т.е. не порождают утечки ресурсов и не нарушают инварианты контейнера при возникновении исключений). Функции-члены `merge()`, `remove()`, `remove_if()` и `unique()` дают гарантии при условии, что сравнение (с помощью оператора == или предиката) не генерирует исключений. Таким образом, используя термин из области баз данных, можно сказать, что списки обеспечивают *безопасные*

транзакции, при условии, что вы не выполняете операции присваивания и функцию-член `sort()` и сравнения не генерируют исключений. В табл. 7.26 перечислены все операции, предоставляющие особые гарантии в отношении исключений. Общее обсуждение обработки исключений в библиотеке STL приведено в разделе 6.12.2.

Таблица 7.26. Операции над списками, предоставляющие специальные гарантии в отношении исключений

Операция	Гарантия
`push_back()`	Либо успешна, либо ничего не делает
`push_front()`	Либо успешна, либо ничего не делает
`insert()`	Либо успешна, либо ничего не делает
`pop_back()`	Не генерирует исключений
`pop_front()`	Не генерирует исключений
`erase()`	Не генерирует исключений
`clear()`	Не генерирует исключений
`resize()`	Либо успешна, либо ничего не делает
`remove()`	Не генерирует исключений, если операция сравнения элементов не генерирует исключений
`remove_if()`	Не генерирует исключений, если предикат не генерирует исключений
`unique()`	Не генерирует исключений, если операция сравнения элементов не генерирует исключений
`splice()`	Не генерирует исключений
`merge()`	Либо успешна, либо ничего не делает, если операция сравнения элементов не генерирует исключений
`reverse()`	Не генерирует исключений
`swap()`	Не генерирует исключений

7.5.4. Примеры использования списков

Следующий пример, в частности, демонстрирует использование специальных функций-членов списков:

```
// cont/list1.cpp

#include <list>
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

void printLists (const list<int>& l1, const list<int>& l2)
{
   cout << "list1: ";
   copy (l1.cbegin(), l1.cend(), ostream_iterator<int>(cout," "));
   cout << endl << "list2: ";
   copy (l2.cbegin(), l2.cend(), ostream_iterator<int>(cout," "));
```

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

int main()
{
   // создаем два пустых списка
   list<int> list1, list2;

   // заполняем оба списка элементами
   for (int i=0; i<6; ++i) {
   list1.push_back(i);
   list2.push_front(i);
   }
   printLists(list1, list2);

   // вставляем все элементы списка list1 перед первым элементом,
   // имеющим значение 3 в списке list2
   // - find() возвращает итератор, установленный на первый элемент,
   // имеющий значение 3
   list2.splice(find(list2.begin(),list2.end(), // позиция назначения
                     3),
                list1);                         // источник
   printLists(list1, list2);

   // перемещает первый элемент списка list2 в конец
   list2.splice(list2.end(),     // позиция назначения
                list2,           // источник
                list2.begin());  // позиция источника
   printLists(list1, list2);

   // сортируем второй список, присваиваем списку list1 и удаляем дубликаты
   list2.sort();
   list1 = list2;
   list2.unique();
   printLists(list1, list2);

   // объединяем оба упорядоченных списка в первый список
   list1.merge(list2);
   printLists(list1, list2);
}
```

Результаты работы программы приведены ниже.

```
list1: 0 1 2 3 4 5
list2: 5 4 3 2 1 0
list1:
list2: 5 4 0 1 2 3 4 5 3 2 1 0
list1:
list2: 4 0 1 2 3 4 5 3 2 1 0 5
list1: 0 0 1 1 2 2 3 3 4 4 5 5
list2: 0 1 2 3 4 5
list1: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
list2:
```

Соответствующий пример использования односвязного списка приведен в разделе 7.6.4.

7.6. Последовательные списки

Последовательный список (экземпляр контейнерного класса `forward_list<>`, введенного в стандарте C++11) управляет своими элементами как односвязный список (рис. 7.8). Как обычно, стандартная библиотека C++ не указывает вид его реализации, но это следует из его названия, ограничений и спецификаций.

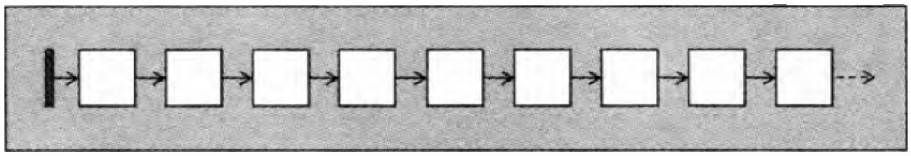


Рис. 7.8. Структура последовательного списка

Для использования последовательного списка необходимо включить заголовочный файл `<forward_list>`.

```
#include <forward_list>
```

В этом файле последовательный список определен как шаблонный класс `T` в пространстве имен `std`.

```
namespace std {
   template <typename T,
             typename Allocator = allocator<T> >
   class forward_list;
}
```

Элементы последовательного списка могут иметь любой тип `T`. Необязательный второй шаблонный параметр определяет модель памяти (см. главу 19). По умолчанию используется модель `allocator`, предусмотренная стандартной библиотекой C++.

7.6.1. Возможности последовательных списков

С теоретической точки зрения последовательный список является списком (объектом класса `list<>`), ограниченным тем, что в нем невозможен обратный обход. В нем нет функций, которых не было бы в обычном списке. В качестве преимущества указывают на то, что он использует меньше памяти и работает немного быстрее. В стандарте утверждается: "*Список типа* `forward_list` *не должен занимать больше памяти и работать медленнее, чем односвязный список, написанный вручную на языке С. Возможности, мешающие этому, должны быть устранены*".

Последовательные списки имеют следующие ограничения по сравнению с обычными списками.

- Последовательный список имеет только однонаправленные, а не двунаправленные итераторы. Как следствие, в нем не предусмотрены обратные итераторы, т.е. такие типы, как `reverse_iterator`, и члены-функции `rbegin()`, `rend()`, `crbegin()` и `crend()`.
- Последовательный список не содержит функции-члена `size()`. Это является следствием игнорирования функциональных возможностей, требующих

дополнительных затрат памяти и времени по сравнению с односвязными списками, написанными вручную.

- Якорь последовательного списка не имеет указателя на последний элемент. По этой причине последовательный список не предусматривает специальных функций-членов для работы с последним элементом, таких как `back()`, `push_back()` и `pop_back()`.
- Для всех функций-членов, модифицирующих последовательные списки путем вставки или удаления элементов на определенных позициях, предусмотрены специальные версии. Причина заключается в том, что в этом случае необходимо передавать позицию элемента, стоящего *перед* первым обрабатываемым элементом, поскольку мы должны присвоить новый следующий за ним элемент. Поскольку обратный проход невозможен (по крайней мере, за константное время), всем этим функциям-членам необходимо передавать позицию предшествующего элемента. Из-за этого отличия имена этих функций-членов имеют суффикс `_after`. Например, вместо функции-члена `insert()` используется функция-член `insert_after()`, вставляющая новые элементы после элемента, переданного в качестве первого аргумента; иначе говоря, она *добавляет* элемент в эту позицию.
- По этой причине последовательные списки имеют функции `before_begin()` и `cbefore_begin()`, возвращающие позицию виртуального элемента, стоящего перед первым элементом (технически — якорь связанного списка), который можно использовать для применения встроенных алгоритмов, имена которых заканчиваются суффиксом `_after`, даже к первым элементам.

Решение не использовать функцию-член `size()` может показаться особенно удивительным, потому что это одна из операций, необходимых для всех контейнеров STL (см. раздел 7.1.2). Это является следствием нежелания тратить дополнительную память и время по сравнению с односвязным списком, написанным вручную на языке C. В качестве альтернативы можно было бы либо вычислять размер последовательного списка каждый раз, когда вызывается функция-член `size()`, что имело бы линейную сложность, либо предусмотреть дополнительное поле для хранения размера в объекте класса `forward_list`, которое обновлялось бы каждый раз при вызове каждой операции, изменяющей количество элементов. В проектной документации последовательного списка сказано [*N2543:FwdList*]: "Это цена, которую должны были бы заплатить все пользователи, независимо от того, нужна ли им эта возможность или нет". Итак, если вам нужен размер последовательного списка, либо отслеживайте его за пределами объекта класса `forward_list`, либо используйте класс `list`.

За исключением этих отличий, последовательные списки работают так же, как обычные списки.

- Последовательный список не предоставляет произвольного доступа. Например, для доступа к пятому элементу необходимо пройти через первые четыре элемента, следуя по цепочке связей. Таким образом, доступ к произвольному элементу списка выполняется медленно.
- Вставка и удаление элементов выполняется быстро на любой позиции (при условии, что итератор уже установлен на эту позицию). Вставка и удаление элементов

всегда выполняется за константное время, потому что никакие другие элементы перемещать не требуется. Все операции сводятся к манипуляциям указателями.

- Вставка и удаление элементов не делает некорректными указатели, ссылки и итераторы, установленные на другие элементы.
- Последовательный список поддерживает обработку исключений таким образом, что почти каждая операция либо выполняется успешно, либо не делает ничего. Таким образом, список не может оказаться в промежуточном положении из-за незавершенной операции.
- Последовательные списки предусматривают много специальных функций-членов для перемещения и удаления элементов. Эти функции-члены работают быстрее общих алгоритмов, потому что они лишь перенаправляют указатели, а не копируют и не перемещают элементы. Однако если используется позиция элемента, функции-члену необходимо передавать предыдущую позицию, а ее имя имеет суффикс `_after`.

7.6.2. Операции над последовательными списками

Создание, копирование и удаление

Возможности создания, копирования и уничтожения последовательных списков не отличаются от аналогичных возможностей любого последовательного контейнера. Операции над последовательными списками перечислены в табл. 7.27. Замечание о возможных источниках инициализации приведены в разделе 7.1.2.

Таблица 7.27. Конструкторы и деструкторы последовательных списков

Операция	Действие
`forward_list <Elem> c`	Конструктор по умолчанию; создает пустой последовательный список, не содержащий ни одного элемента
`forward_list <Elem> c (c2)`	Копирующий конструктор; создает новый последовательный список как копию последовательного списка *c2* (копируются все элементы)
`forward_list <Elem> c = c2`	Копирующий конструктор; создает новый последовательный список как копию последовательного списка *c2* (копируются все элементы)
`forward_list <Elem> c (rv)`	Перемещающий конструктор; создает новый последовательный список, получающий содержимое последовательного списка *rv* (по стандарту C++11)
`forward_list <Elem> c = rv`	Перемещающий конструктор; создает новый последовательный список, получающий содержимое последовательного списка *rv* (по стандарту C++11)

Окончание табл. 7.27

Операция	Действие
`forward_list <Elem> c(n)`	Создает последовательный список из *n* элементов, созданных конструктором по умолчанию
`forward_list <Elem> c(n,elem)`	Создает последовательный список, инициализированный *n* копиями элемента *elem*
`forward_list <Elem> c(beg,end)`	Создает последовательный список, инициализированный элементами интервала [*beg*,*end*)
`forward_list <Elem> c(initlist)`	Создает последовательный список, инициализированный элементами последовательного списка инициализации *initlist* (по стандарту C++11)
`forward_list <Elem> c = initlist`	Создает последовательный список, инициализированный элементами последовательного списка инициализации *initlist* (по стандарту C++11)
`c.~forward_list()`	Уничтожает все элементы и освобождает память

Немодифицирующие операции

За одним исключением, последовательные списки предусматривают обычные операции для работы с размером и сравнений: у последовательных списков нет операции `size()`. Причина заключается в том, что за константное время невозможно ни вычислить, ни сохранить текущее количество элементов. Для того чтобы подчеркнуть, что функция `size()` является затратной операций, ее не включили в класс `forward_list<>`. Если возникает необходимость вычислить количество элементов последовательного списка, можно воспользоваться алгоритмом `distance()` (см. раздел 9.3.3).

```
#include <forward_list>
#include <iterator>
std::forward_list<int> l;
...
std::cout << "l.size(): " << std::distance(l.begin(),l.end())
          << std::endl;
```

Однако следует иметь в виду, что алгоритм `distance()` в этой программе выполняется за линейное время.

Полный список немодифицирующих операций над последовательными списками приведен в табл. 7.28, а детали остальных операций описаны в разделе 7.1.2.

Таблица 7.28. Немодифицирующие операции над последовательными списками

Операция	Действие
`c.empty()`	Возвращает признак того, что контейнер пуст
`c.max_size()`	Возвращает максимально возможное количество элементов
c1 == *c2*	Возвращает результат проверки того, что последовательный список *c1* равен последовательному списку *c2* (выполняет оператор == для элементов)

Окончание табл. 7.28

Операция	Действие
c1 != *c2*	Возвращает результат проверки того, что последовательный список *c1* не равен последовательному списку *c2* (эквивалент ! (*c1*==*c2*))
c1 < *c2*	Возвращает результат проверки того, что последовательный список *c1* меньше последовательного списка *c2*
c1 > *c2*	Возвращает результат проверки того, что последовательный список *c1* больше последовательного списка *c2* (эквивалент *c2*<*c1*)
c1 <= *c2*	Возвращает результат проверки того, что последовательный список *c1* не больше последовательного списка *c2* (эквивалент ! (*c2*<*c1*))
c1 >= *c2*	Возвращает результат проверки того, что последовательный список *c1* не меньше последовательного списка *c2* (эквивалент ! (*c1*<*c2*))

Операции присваивания

Последовательные списки также предусматривают операции присваивания, типичные для последовательных контейнеров (табл. 7.29). Как обычно, операции вставки соответствуют конструкторам для обеспечения разных источников инициализации (см. раздел 7.1.2).

Таблица 7.29. Операции присваивания для последовательных списков

Операция	Действие
c = *c2*	Присваивание всех элементов последовательного списка *c2* последовательному списку *c*
c = *rv*	Перемещение всех элементов последовательного списка *rv* в последовательный список *c* (по стандарту C++11)
c = *initlist*	Присваивание всех элементов последовательного списка инициализации *initlist* последовательному списку *c* (по стандарту C++11)
c.assign(*n*, *elem*)	Присваивание *n* копий элемента *elem*
c.assign(*beg*, *end*)	Присваивание элементов интервала [*beg*,*end*)
c.assign(*initlist*)	Присваивание всех элементов последовательного списка инициализации *initlist* последовательному списку *c*
c1.swap(*c2*)	Обмен данными между последовательными списками *c1* и *c2*
swap(*c1*,*c2*)	Обмен данными между последовательными списками *c1* и *c2*

Доступ к элементам

Для доступа ко всем элементам списка необходимо использовать диапазонные циклы `for` (см. раздел 3.1.4), специальные операции или итераторы. В отличие от списков, последовательные списки поддерживают прямой доступ только к первому элементу, и поэтому в них предусмотрена только функция-член `front()`, предоставляющая прямой доступ к первому элементу (табл. 7.30).

Таблица 7.30. Прямой доступ к элементам последовательного списка

Операция	Действие
`c.front()`	Возвращает первый элемент (*не* проверяя, существует ли этот элемент)

Как обычно, эта операция *не* проверяет, пуст ли контейнер. Если контейнер пуст, вызов этой операции приводит к непредсказуемым последствиям. Таким образом, вызывающая сторона должна гарантировать, что контейнер содержит хотя бы один элемент. Кроме того, в многопотоковой среде необходимы механизмы синхронизации, гарантирующие, что список не будет изменен между проверкой его размера и выполнением операции доступа к элементу (см. раздел 18.4.3).

Функции для итераторов

Для доступа ко всем элементам последовательного списка необходимо использовать итераторы. Однако, поскольку последовательный список обеспечивает только прямой обход, его итераторы могут быть только однонаправленными, а обратные итераторы не поддерживаются (табл. 7.31).

Таким образом, к спискам нельзя применять алгоритмы, требующие итераторы произвольного доступа. К этой категории относятся все алгоритмы, интенсивно оперирующие порядком элементов, особенно алгоритмы сортировки. Однако для сортировки элементов списки содержат специальную функцию-член `sort()` (см. раздел 8.8.1).

Кроме того, в классе `forward_list<>` предусмотрены функции-члены `before_begin()` и `cbefore_begin()` для определения позиции виртуального элемента, занимающего позицию перед первым элементом. Это необходимо для модификации следующего элемента, если этим элементом является первый элемент.

Таблица 7.31. Функции для работы с итераторами списка

Операция	Действие
`c.begin()`	Возвращает однонаправленный итератор, установленный на первый элемент
`c.end()`	Возвращает однонаправленный итератор, установленный на позицию, следующую за последним элементом
`c.cbegin()`	Возвращает константный однонаправленный итератор, установленный на первый элемент (по стандарту C++11)
`c.cend()`	Возвращает константный однонаправленный итератор, установленный на позицию, следующую за последним элементом (начиная со стандарта C++11)
`c.before_begin()`	Возвращает однонаправленный итератор, установленный на позицию, предшествующую первому элементу
`c.cbefore_begin()`	Возвращает константный однонаправленный итератор, установленный на позицию, предшествующую первому элементу

Отметим, что итераторы, возвращаемые функциями-членами `before_begin()` и `cbefore_begin()`, не соответствуют корректным позициям в последовательном списке. Следовательно, их разыменование приводит к непредсказуемым последствиям. Таким

образом, использование любого алгоритма, которому в качестве первого аргумента передается функция-член `before_begin()`, приводит к ошибке времени выполнения программы.

```
// ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ: функцию-член before_begin() можно применять
// только в сочетании с операциями ..._after()
std::copy (fwlist.before_begin(), fwlist.end(),...);
```

Помимо копирования и присваивания единственными корректными операциями над значениями, возвращаемыми функцией-членом `before_begin()`, являются `++`, `==` и `!=`.

Вставка и удаление элементов

В табл. 7.32 перечислены операции вставки и удаления элементов последовательных списков. Из-за природы списков в целом и последовательных списков в частности их следует рассмотреть подробно.

Таблица 7.32. Вставка и удаление элементов последовательных списков

Операция	Действие
`c.push_front(`*elem*`)`	Вставляет копию аргумента *elem* в начало последовательного списка
`c.pop_front()`	Удаляет первый элемент (не возвращая его)
`c.insert_after(`*pos*`,`*elem*`)`	Вставляет копию аргумента *elem* после позиции итератора *pos* и возвращает позицию нового элемента
`c.insert_after(`*pos*`,`*n*`,`*elem*`)`	Вставляет *n* копий аргумента *elem* после позиции итератора *pos*, и возвращает позицию первого нового элемента (или *pos*, если нового элемента нет)
`c.insert_after(`*pos*`,`*beg*`,`*end*`)`	Вставляет копии всех элементов интервала [*beg*,*end*) после позиции итератора *pos* и возвращает первый новый элемент (или позицию *pos*, если нового элемента нет)
`c.insert_after(`*pos*`,`*initlist*`)`	Вставляет копии всех элементов списка инициализации *initlist* после позиции итератора *pos* и возвращает позицию первого нового элемента (или позицию *pos*, если нового элемента нет; по стандарту C++11)
`c.emplace_after(`*pos*`,`*args*...`)`	Вставляет новый элемент, инициализированный списком аргументов args после позиции итератора *pos* и возвращает позицию нового элемента (по стандарту C++11)
`c.emplace_front(`*args*`...)`	Добавляет в начало списка новый элемент, инициализированный списком аргументов args (ничего не возвращая; по стандарту C++11)
`c.erase_after(`*pos*`)`	Удаляет элемент, следующий за позицией итератора *pos* (ничего не возвращая)
`c.erase_after(`*beg*`,`*end*`)`	Удаляет все элементы интервала (*beg*,*end*) (ничего не возвращая)

Окончание табл. 7.32

Операция	Действие
`c.remove(`*val*`)`	Удаляет все элементы, имеющие значение `val`
`c.remove_if(`*op*`)`	Удаляет все элементы, для которых операция *op*`(elem)` возвращает `true`
`c.resize(`*num*`)`	Изменяет количество элементов до *num* (если размер увеличивается, новые элементы создаются их конструкторами по умолчанию)
`c.resize(`*num*`,`*elem*`)`	Изменяет количество элементов на *num* (если размер увеличивается, новые элементы являются копиями объекта *elem*)
`c.clear()`	Удаляет все элементы (опустошает контейнер)

На последовательные списки распространяются ограничения, характерные для списков вообще.

- Как обычно при использовании библиотеки STL, программист должен гарантировать, что аргументы являются корректными. Итераторы должны ссылаться на корректные позиции, а начало интервала должно предшествовать его концу.
- Вставка и удаление выполняются быстрее, если при работе с несколькими элементами мы используем один вызов операции для нескольких элементов, а не несколько вызовов.

Как и при работе с обычными списками, для удаления элементов последовательные списки предусматривают специальные реализации алгоритмов `remove()` (см. раздел 11.7.1). Эти функции-члены работают быстрее, чем алгоритмы `remove()`, потому что они манипулируют только внутренними указателями, а не элементами. Более подробное описание операций над списками приведено в разделе 7.5.2.

Отметим, что при работе со всеми функциями-членами последовательных списков, предназначенными для вставки, перестановки и очистки, возникает одна и та же проблема: они обычно получают позицию элемента, предназначенную для вставки или удаления. Однако это требует модификации *предшествующего* элемента, потому что необходимо изменить его указатель, ссылающийся на следующий элемент. В обычных списках можно просто вернуться к предыдущему элементу, но в последовательных списках это невозможно. По этой причине упомянутые функции-члены работаю иначе, чем в обычных списках. Этот факт отражен в их именах. Все они имеют суффикс `_after`, т.е. они вставляют новый элемент *после* передаваемого элемента (иначе говоря, они добавляют *элемент*) или удаляют элемент, размещенный *после* передаваемого элемента.

При использовании указанных функций-членов в сочетании с функцией-членом `before_begin()` обеспечить корректность первого элемента последовательного списка и получить прямой доступ к нему можно следующим образом (рис. 7.9):

```
std::forward_list<int> fwlist = { 1, 2, 3 };

// вставляем в начало 77, 88 и 99:
fwlist.insert_after (fwlist.before_begin(), // позиция
                     { 77, 88, 99 } );      // значения
```

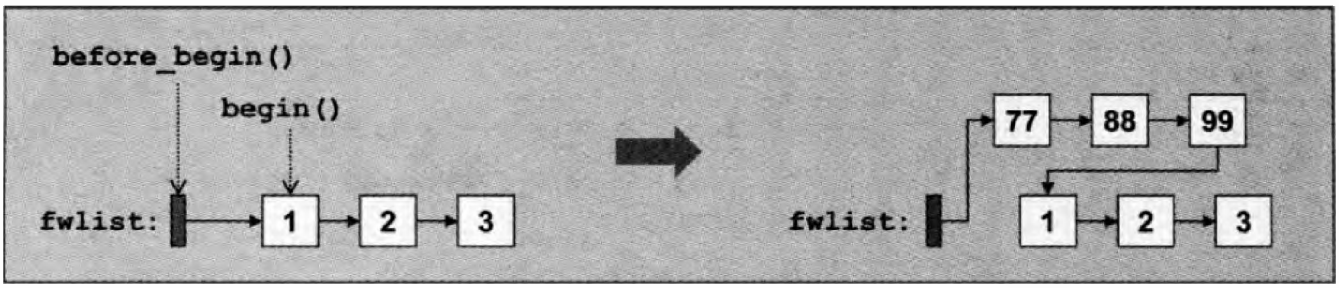


Рис. 7.9. Вставка элементов в начало последовательного списка

Отметим, что применение функции-члена с именем, заканчивающимся суффиксом `_after`, к итераторам `end()` и `cend()` приводит к непредсказуемым последствиям, так как для добавления нового элемента в конец форвардного списка необходимо передать позицию последнего элемента (или итератор `before_begin()`, если его нет).

```
// ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ПРОГРАММЫ: добавление элемента в позицию,
// расположенную за концом последовательного списка,
// приводит к непредсказуемым последствиям
fwlist.insert_after(fwlist.end(),9999);
```

Поиск и удаление или вставка

Недостатки односвязных списков, допускающих только прямой обход, лишь усугубляются при попытке найти элемент для вставки или удаления. Проблема заключается в том, что когда вы найдете элемент, то окажетесь слишком далеко, потому что для вставки или удаления необходимо изменить предшествующий элемент. По этой причине необходимо найти элемент, *следующий* за которым удовлетворяет заданному критерию. Рассмотрим пример:

```
// cont/forwardlistfind1.cpp

#include <forward_list>
#include "print.hpp"
using namespace std;

int main()
{
   forward_list<int> list = { 1, 2, 3, 4, 5, 97, 98, 99 };

   // находим позицию перед первым четным элементом
   auto posBefore = list.before_begin();
   for (auto pos=list.begin(); pos!=list.end(); ++pos, ++posBefore) {
        if (*pos % 2 == 0) {
            break; // элемент найден
        }
   }

   // вставляем новый элемент перед первым четным элементом
   list.insert_after(posBefore,42);
   PRINT_ELEMENTS(list);
}
```

Здесь итератор pos проходит по списку в ходе поиска заданного элемента, а итератор posBefore всегда расположен перед итератором pos, чтобы можно было вернуть позицию элемента, расположенного перед искомым (см. рис. 7.10). Результат программы выглядит следующим образом:

```
1 42 2 3 4 5 97 98 99
```

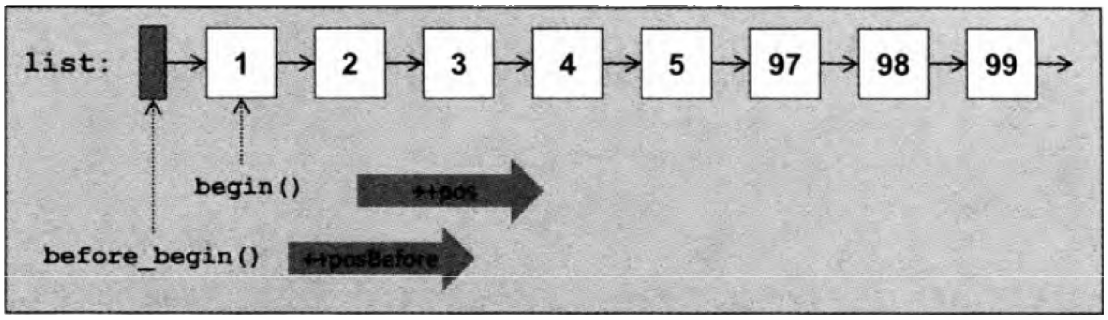


Рис. 7.10. Поиск позиции для вставки или удаления

В качестве альтернативы можно использовать удобную функцию-член next() для работы с итераторами, которая стала доступной в стандарте C++11 (см. раздел 9.3.2).

```
#include <iterator>
...
auto posBefore = list.before_begin();
for ( ; next(posBefore)!=list.end(); ++posBefore) {
    if (*next(posBefore) % 2 == 0) {
        break; // элемент найден
    }
}
```

Если эту операцию приходится выполнять достаточно часто, можно определить собственный алгоритм для поиска позиции перед элементом, имеющим заданное значение или удовлетворяющим конкретному критерию.

```
// cont/findbefore.hpp

template <typename InputIterator, typename Tp>
inline InputIterator
find_before (InputIterator first, InputIterator last, const Tp& val)
{
   if (first==last) {
       return first;
   }
   InputIterator next(first);
   ++next;
   while (next!=last && !(*next==val)) {
      ++next;
      ++first;
   }
   return first;
}

template <typename InputIterator, typename Pred>
```

```
inline InputIterator
find_before_if (InputIterator first, InputIterator last, Pred pred)
{
   if (first==last) {
       return first;
   }
   InputIterator next(first);
   ++next;
   while (next!=last && !pred(*next)) {
      ++next;
      ++first;
   }
   return first;
}
```

Работая с этими алгоритмами, для поиска соответствующей позиции можно использовать лямбда-функции (законченный пример приведен в файле `cont/fwlistfind2.cpp`).

```
// находим позицию перед первым четным элементом
auto posBefore = find_before_if (list.before_begin(), list.end(),
                                 [] (int i) {
                                 return i%2==0;
                                 });
// вставляем новый элемент перед ним
list.insert_after(posBefore,42);
```

Мы должны вызвать функцию-член `find_before_if()` с позицией, возвращаемой функцией-членом `before_begin()`. В противном случае мы пропустим первый элемент. Для того чтобы избежать непредсказуемых последствий при передаче позиции, возвращаемой функцией-членом `begin()`, алгоритмы сначала проверяют, не совпадают ли начало и конец интервала. Было бы лучше включить в последовательные списки аналогичные функции-члены, но, к сожалению, это не было сделано.

Функции срезки и функции, изменяющие порядок элементов

Как и обычные списки, последовательные списки имеют одно преимущество: удаление и вставка элементов в любой позиции выполняются за константное время. При переносе элементов из одного контейнера в другой это преимущество усиливается, потому что для этого достаточно перенаправить внутренние указатели. По этой причине последовательные списки имеют почти те же самые функции-члены для срезки и изменения порядка следования элементов, что и обычные списки. Эти операции можно выполнять для перемещения элементов в списке или между двумя списками, при условии, что списки имеют одинаковые типы. Единственным отличием от списков является то, что в классе определена функция-член `splice_after()`, а не `splice()`, потому что ей передается позиция элемента, расположенного перед позицией срезки.

Эти функции перечислены в табл. 7.33. Они подробно описаны в разделе 8.8. Следующая программа демонстрирует, как использовать функции срезки для последовательных списков. Здесь первый элемент со значением `3` в последовательном списке `l1` перемещается на позицию перед первым элементом, имеющим значение `99` в последовательном списке `l2`.

```
// cont/forwardlistsplice1.cpp

#include <forward_list>
```

```
#include "print.hpp"
using namespace std;

int main()
{
   forward_list<int> l1 = { 1, 2, 3, 4, 5 };
   forward_list<int> l2 = { 97, 98, 99 };

   // ищем 3 в последовательном списке l1
   auto pos1=l1.before_begin();
   for (auto pb1=l1.begin(); pb1 != l1.end(); ++pb1, ++pos1) {
        if (*pb1 == 3) {
            break; // Найден
        }
    }

   // ищем 99 в последовательном списке l2
   auto pos2=l2.before_begin();
   for (auto pb2=l2.begin(); pb2 != l2.end(); ++pb2, ++pos2) {
        if (*pb2 == 99) {
            break; // Найден
        }
   }

   // срезка от 3 до l1 в последовательный список l2 перед значением 99
   l1.splice_after(pos2, l2, // назначение
                   pos1);    // источник

   PRINT_ELEMENTS(l1,"l1: ");
   PRINT_ELEMENTS(l2,"l2: ");
}
```

Таблица 7.33. Специальные модифицирующие операции над последовательными списками

Операция	Действие
c.unique()	Удаляет дубликаты последовательных элементов, имеющих одинаковые значения
c.unique(*op*)	Удаляет дубликаты последовательных элементов, для которых операция *op*() возвращает true
c.splice_after(*pos*,*c2*)	Перемещает все элементы последовательного списка *c2* в последовательный список *c* и размещает их сразу за позицией итератора *pos*
c.splice_after(*pos*,*c2*,*c2pos*)	Перемещает элемент, занимающий позицию за *c2pos* в последовательном списке *c2*, в последовательный список *c* и размещает его сразу за позицией *pos* (списки *c* и *c2* могут быть идентичными)

Окончание табл. 7.33

Операция	Действие
`c.splice_after(pos,c2,c2beg,c2end)`	Перемещает все элементы между *c2beg* и *c2end* (не включая концы интервала) последовательного списка *c2* в последовательный список *c* и размещает их сразу за позицией *pos* списка *c* (списки *c* и *c2* могут быть идентичными)
`c.sort()`	Упорядочивает все элементы с помощью оператора <
`c.sort(op)`	Упорядочивает все элементы с помощью операции *op*()
`c.merge(c2)`	В предположении, что оба контейнера содержат упорядоченные элементы, перемещает все элементы списка *c2* в список *c*, так что все элементы сливаются и остаются упорядоченными
`c.merge(c2,op)`	В предположении, что оба контейнера содержат упорядоченные элементы в соответствии с критерием *op*(), перемещает все элементы последовательного списка *c2* в последовательный список *c*, так что все элементы сливаются и остаются упорядоченными в соответствии с критерием *op*()
`c.reverse()`	Изменяет на противоположный порядок следования всех элементов

Сначала в последовательном списке `l1` мы находим позицию перед первым элементом со значением `3`. Затем в последовательном списке `l2` мы ищем позицию перед первым элементом, имеющим значение `99`. В заключение обе позиции передаются функции-члену `splice_after()`, которая просто модифицирует внутренние указатели в списке (рис. 7.11).

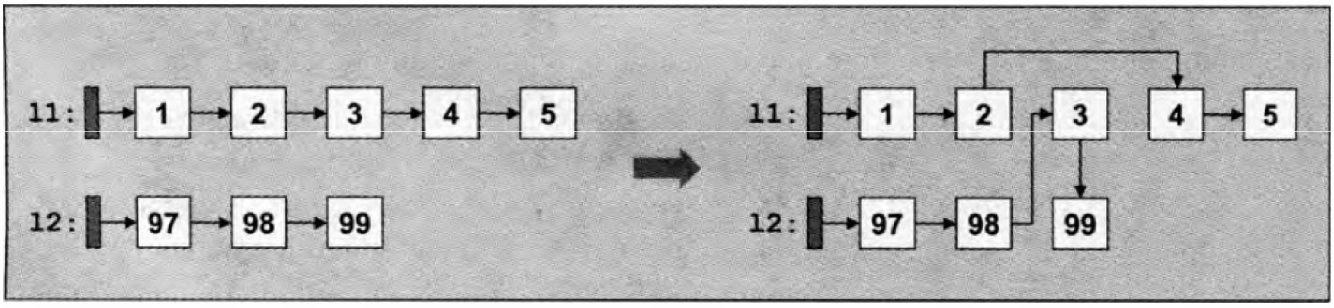


Рис. 7.11. Действие функции ***splice_after()*** *на последовательный список*

Алгоритм `find_before()` может упростить программу.

```
// срезка от 3 до l1 в последовательный список l2 перед значением 99
l1.splice_after(l2.find_before(99), l2, // назначение
                l1.find_before(3));     // источник
```

Отметим, что источник и назначение для операции срезки могут совпадать. Таким образом, можно перемещать элементы в последовательном списке. Однако следует подчеркнуть, что применение функции-члена `splice_after()` к позиции, возвращаемой функцией-членом `end()`, приводит к непредсказуемым последствиям, как и все функции, имена которых заканчиваются суффиксом `_after`, получающие позицию от функции-члена `end()`.

```
// ОШИБКА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ПРОГРАММЫ: такое перемещение первого элемента
// в конец списка невозможно
fwlist.splice_after(fwlist.end(),    // позиция назначения
                    fwlist,          // список-источник
                    fwlist.begin()); // позиция в источнике
```

7.6.3. Обработка исключений

Последовательные списки предоставляют те же гарантии относительно исключений, что и обычные списки, при условии, что соответствующая функция-член является доступной. Детали см. в разделе 7.5.3.

7.6.4. Примеры использования последовательных списков

Следующий пример демонстрирует использование специальных функций-членов последовательных списков:

```
// cont/forwardlist1.cpp

#include <forward_list>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <string>
using namespace std;

void printLists (const string& s, const forward_list<int>& l1,
                 const forward_list<int>& l2)
{
   cout << s << endl;
   cout << " list1: ";
   copy (l1.cbegin(), l1.cend(), ostream_iterator<int>(cout," "));
   cout << endl << " list2: ";
   copy (l2.cbegin(), l2.cend(), ostream_iterator<int>(cout," "));
   cout << endl;
}

int main()
{
   // создаем два последовательных списка
   forward_list<int> list1 = { 1, 2, 3, 4 };
   forward_list<int> list2 = { 77, 88, 99 };
   printLists ("initial:", list1, list2);

   // вставляем шесть новых элементов в начало последовательного списка list2
   list2.insert_after(list2.before_begin(),99);
```

```
    list2.push_front(10);
    list2.insert_after(list2.before_begin(), {10,11,12,13} );
    printLists ("6 new elems:", list1, list2);

    // вставляем все элементы последовательного списка list2
    // в начало последовательного списка list1
    list1.insert_after(list1.before_begin(),
    list2.begin(),list2.end());
    printLists ("list2 into list1:", list1, list2);

    // удаляем второй элемент и элементы, расположенные после элемента,
    // имеющего значение 99
    list2.erase_after(list2.begin());
    list2.erase_after(find(list2.begin(), list2.end(), 99), list2.end());
    printLists ("delete 2nd and after 99:", list1, list2);

    // сортируем последовательный список list1,
    // присваиваем его последовательному списку list2 и
    // удаляем дубликаты
    list1.sort();
    list2 = list1;
    list2.unique();
    printLists ("sorted and unique:", list1, list2);

    // объединяем упорядоченные списки в последовательном списке list1
    list1.merge(list2);
    printLists ("merged:", list1, list2);
}
```

Результаты работы программы выглядят следующим образом:

```
initial:
  list1: 1 2 3 4
  list2: 77 88 99
6 new elems:
  list1: 1 2 3 4
  list2: 10 11 12 13 10 99 77 88 99
list2 into list1:
  list1: 10 11 12 13 10 99 77 88 99 1 2 3 4
  list2: 10 11 12 13 10 99 77 88 99
delete 2nd and after 99:
  list1: 10 11 12 13 10 99 77 88 99 1 2 3 4
  list2: 10 12 13 10 99
sorted and unique:
  list1: 1 2 3 4 10 10 11 12 13 77 88 99 99
  list2: 1 2 3 4 10 11 12 13 77 88 99
merged:
  list1: 1 1 2 2 3 3 4 4 10 10 10 11 11 12 12 13 13 77 77 88 88 99 99 99
  list2:
```

Соответствующий пример, демонстрирующий работу с обычным списком, приведен в разделе 7.5.4.

7.7. Множества и мультимножества

Множества и мультимножества автоматически упорядочивают свои элементы в соответствии с заданным критерием. Разница между ними заключается в том, что мультимножества допускают дубликаты, а множества нет (рис. 7.12), как было указано в главе 6.

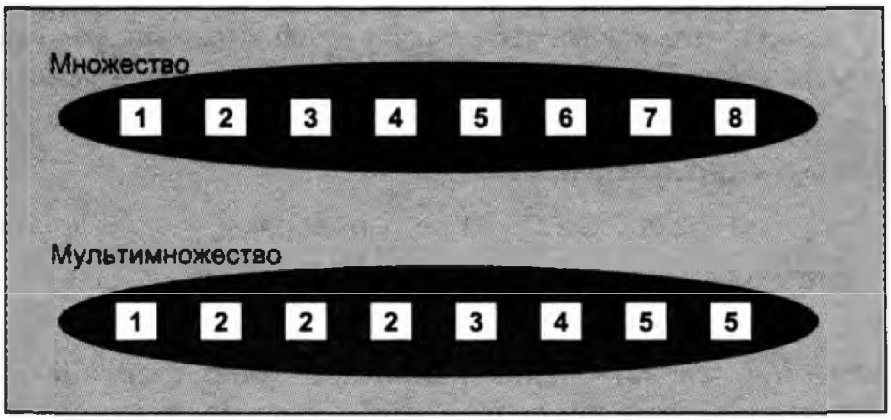


Рис. 7.12. Множества и мультимножества

Для использования множества или мультимножества необходимо включить в программу заголовочный файл `<set>`.

```
#include <set>
```

Здесь эти типы определены как шаблонные классы в пространстве имен `std`.

```
namespace std {
   template <typename T,
             typename Compare = less<T>,
             typename Allocator = allocator<T> >
             class set;

   template <typename T,
             typename Compare = less<T>,
             typename Allocator = allocator<T> >
             class multiset;
}
```

Элементы множества или мультимножества могут иметь любой тип `T`, совместимый с критерием сортировки, который определяется необязательным вторым шаблонным параметром. Если конкретный критерий сортировки не указан, то по умолчанию используется критерий `less`. Функциональный объект `less` упорядочивает элементы, сравнивая их с помощью оператора < (см. раздел 10.2.1). Необязательный третий шаблонный параметр задает модель памяти (см. главу 19). По умолчанию используется модель памяти `allocator`, предусмотренная стандартной библиотекой C++.

Критерий сортировки должен определять *строгое слабое упорядочение* (strict weak ordering), определяемое следующими четырьмя свойствами.

1. Оно должно быть **антисимметричным**.
2. Для оператора < это значит, что если выражение `x < y` имеет значение `true`, то выражение `y < x` имеет значение `false`.

3. Для предиката `op()` это значит, что если выражение `op(x,y)` имеет значение `true`, то выражение `op(y,x)` имеет значение `false`.
4. Оно должно быть **транзитивным**.
5. Для операции < это значит, что если значение выражения `x < y` равно `true` и значение выражения `y < z` равно `true`, то значение выражения `x < z` равно `true`.
6. Для предиката `op()` это значит, что если значение выражения `op(x,y)` равно `true` и значение выражения `op(y,z)` равно `true`, то значение выражения `op(x,z)` равно `true`.
7. Оно должно быть **иррефлексивным**.
8. Для операции < это значит, что значение выражения `x < x` всегда равно `false`.
9. Для предиката `op()` это значит, что значение выражения `op(x,x)` всегда равно `false`.
10. Оно должно иметь свойство **транзитивности эквивалентности** (transitivity of equivalence), которое, грубо говоря, означает следующее: если объект `a` эквивалентен объекту `b`, а объект `b` эквивалентен объекту `c`, то объект `a` эквивалентен объекту `c`.
11. Для оператора < это значит, что если значение выражения `!(a<b) && !(b<a)` равно `true` и значение выражения `!(b<c) && !(c<b)` равно `true`, то значение выражения `!(a<c) && !(c<a)` равно `true`.
12. Для предиката `op()` это значит, что если значения выражений `op(a,b)`, `op(b,a)`, `op(b,c)` и `op(c,b)` равны `false`, то значения выражений `op(a,c)` и `op(c,a)` равны `false`.

Это значит, что следует различать отношения "меньше" и "равно". Оператор <= не удовлетворяет этому требованию.

Основываясь на этих требованиях, можно использовать критерий сортировки и для проверки эквивалентности. Иначе говоря, два элемента считаются дубликатами, если ни один из них не меньше другого (или оба выражения `op(x,y)` и `op(y,x)` равны `false`).

Для мультимножеств порядок эквивалентных элементов является случайным, но устойчивым. Таким образом, вставки и удаление сохраняют относительный порядок следования эквивалентных элементов (это гарантируется стандартом C++11).

7.7.1. Возможности множеств и мультимножеств

Как и все стандартные ассоциативные контейнеры, множества и мультимножества обычно реализуются в виде сбалансированных бинарных деревьев (рис. 7.13). Стандарт это условие не оговаривает, но оно следует из сложности операций над множествами и мультимножествами[9].

[9] На практике множества и мультимножества обычно реализуются в виде *красно-черных деревьев*, очень удобных для изменения количества и поиска элементов. Они гарантируют, что при вставке произойдет не больше двух изменений связей и что самый длинный путь не превысит более чем вдвое кратчайший путь к листу.

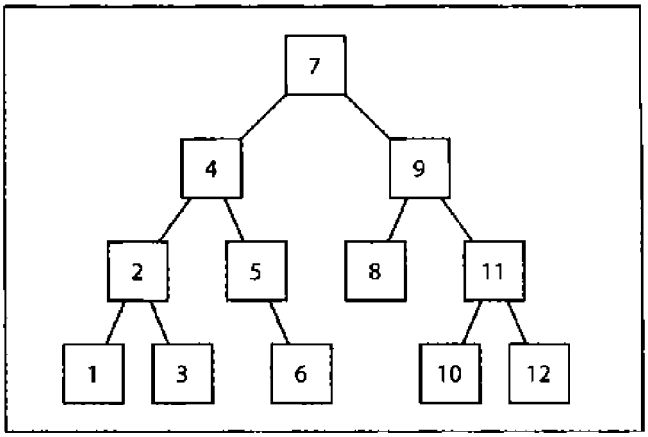


Рис. 7.13. Внутренняя структура множества и мультимножества

Основное преимущество автоматической сортировки состоит в том, что бинарное дерево допускает эффективный поиск конкретного значения. Фактически функции поиска имеют логарифмическую сложность. Например, для поиска элемента в множестве или мультимножестве, состоящем из 1000 элементов, поиск по дереву, осуществляемый функцией-членом, в среднем требует в пятьдесят раз меньше сравнений, чем при линейном поиске (который осуществляется алгоритмами поиска, обходящими все элементы). Более подробно вопросы, касающиеся сложности, рассмотрены в разделе 2.2.

В то же время автоматическая сортировка накладывает важное ограничение на множества и мультимножества: значение элемента *нельзя* изменить непосредственно, потому что это может нарушить правильный порядок следования элементов.

Таким образом, для модификации значения элемента следует удалить элемент, имеющий старое значение, и вставить новый элемент, имеющий новое значение. Эта особенность отражена в интерфейсе.

- Множества и мультимножества не поддерживают прямой доступ к элементам.
- Косвенный доступ через итераторы ограничен тем, что с точки зрения итератора значение элемента является константным.

7.7.2. Операции над множествами и мультимножествами

Создание, копирование и удаление

Конструкторы и деструкторы множеств и мультимножеств перечислены в табл. 7.34.

Таблица 7.34. Конструкторы и деструкторы множеств и мультимножеств

Операция	Действие
set c	Конструктор по умолчанию; создает пустое множество/мультимножество, не содержащее никаких элементов
set c (op)	Создает пустое множество/мультимножество, использующее операцию *op* в качестве критерия сортировки
set c (c2)	Копирующий конструктор; создает копию другого множества/мультимножества того же типа (все элементы копируются)

Окончание табл. 7.34

Операция	Действие
set *c = c2*	Копирующий конструктор; создает копию другого множества/мультимножества того же типа (все элементы копируются)
set *c (rv)*	Перемещающий конструктор; создает новое множество/мультимножество того же типа, получающее содержимое контейнера *rv* (по стандарту C++11)
set *c = rv*	Перемещающий конструктор; создает новое множество/мультимножество того же типа, получающее содержимое контейнера *rv* (по стандарту C++11)
set *c (beg, end)*	Создает множество/мультимножество, инициализированное элементами интервала [*beg*,*end*)
set *c (beg, end, op)*	Создает множество/мультимножество с критерием сортировки *op*, инициализированное элементами интервала [*beg*,*end*)
set *c (initlist)*	Создает множество/мультимножество, инициализированное элементами списка инициализации *initlist* (по стандарту C++11)
set *c = initlist*	Создает множество/мультимножество, инициализированное элементами списка инициализации *initlist* (по стандарту C++11)
*c.~**set**()*	Уничтожает все элементы и освобождает память

Здесь слово ***set*** может означать один из следующих типов.

set	Действие
`set<`*Elem*`>`	Множество, по умолчанию упорядочивающее элементы с помощью критерия `less<>` (оператор <)
`set<`*Elem*`,` *Op*`>`	Множество, по умолчанию упорядочивающее элементы с помощью предиката *Op*
`multiset<`*Elem*`>`	Мультимножество, по умолчанию упорядочивающее элементы с помощью критерия `less<>` (оператор <)
`multiset<`*Elem*`,` *Op*`>`	Мультимножество, по умолчанию упорядочивающее элементы с помощью предиката *Op*

Критерий сортировки можно задать двумя способами.

1. **Как шаблонный параметр**. Например:

```
std::set<int,std::greater<int>> coll;
```

В этом случае критерий сортировки является частью типа. При этом система типов допускает сочетание только контейнеров с одинаковыми критериями сортировки. Это обычный способ задания критерия сортировки. Точнее говоря, второй параметр — эти *тип* критерия сортировки.

Конкретный критерий сортировки представляет собой функциональный объект, созданный контейнером. Для этого конструктор контейнера вызывает конструктор, предусмотренный по умолчанию для типа критерия сортировки. Пример использования пользовательского критерия сортировки приведен в разделе 10.1.1.

2. **Как параметр конструктора.** В этом случае вы можете иметь тип для нескольких критериев сортировки, допускающих разные начальные значения или состояния. Это полезно при обработке критерия сортировки во время выполнения программы, а также в ситуациях, когда критерии сортировки должны быть разными, но иметь одинаковые типы. Законченный пример приведен в разделе 7.7.5.

Если не указан никакой конкретный критерий сортировки, используется функциональный объект `less<>`, заданный по умолчанию и упорядочивающий элементы с помощью оператора <.

Отметим, что критерий сортировки также используется для проверки эквивалентности двух элементов в одном и том же контейнере (т.е. для поиска дубликатов). Таким образом, при использовании критерия сортировки, заданного по умолчанию, проверка эквивалентности двух элементов выглядит примерно так:

```
if (! (elem1<elem2 || elem2<elem1))
```

Это дает три преимущества.

1. В качестве критерия сортировки достаточно передавать только один аргумент.
2. Для типа элемента не обязательно предусматривать оператор ==.
3. Эквивалентность и равенство можно определять по-разному (однако это может стать источником недоразумений).

Проверка эквивалентности указанным способом может занять немного больше времени, потому что для вычисления предыдущего выражения могут потребоваться два сравнения. Впрочем, если результат первого сравнения равен `true`, то второе сравнение не выполняется.

Кроме того, если два контейнера сравниваются с помощью оператора ==, то элементы обоих контейнеров сравниваются с помощью их операторов ==, т.е. тип элементов должен предоставлять оператор ==.

Для инициализации контейнера элементами контейнера, имеющего другой тип, из массива или стандартного потока можно использовать конструктор с указанием начала и конца интервала. Детали см. в разделе 7.1.2.

Немодифицирующие операции

Множества и мультимножества имеют обычные немодифицирующие операции для работы с размером и выполнения сравнений (табл. 7.35).

Таблица 7.35. Немодифицирующие операции над множествами и мультимножествами

Операция	Действие
`c.key_comp()`	Возвращает критерий сравнения
`c.value_comp()`	Возвращает критерий сравнения для значений в целом (эквивалент `key_comp()`)
`c.empty()`	Возвращает результат проверки того, что контейнер пуст (эквивалент выражения `size()==0`, но может работать быстрее)

Окончание табл. 7.35

Операция	Действие
`c.size()`	Возвращает текущее количество элементов
`c.max_size()`	Возвращает максимально возможное количество элементов
c1 == *c2*	Возвращает результат проверки того, что контейнер *c1* равен контейнеру *c2*
c1 != *c2*	Возвращает результат проверки того, что контейнер *c1* не равен контейнеру *c2* (эквивалент ! (c1==c2))
c1 < *c2*	Возвращает результат проверки того, что контейнер *c1* меньше контейнера *c2* (это не относится к неупорядоченным контейнерам)
c1 > *c2*	Возвращает результат проверки того, что контейнер *c1* больше контейнера *c2* (эквивалент c2<c1)
c1 <= *c2*	Возвращает результат проверки того, что контейнер *c1* меньше или равен контейнеру *c2* (эквивалент ! (c2<c1))
c1 >= *c2*	Возвращает результат проверки того, что контейнер *c1* больше или равен контейнеру *c2* (эквивалент ! (c1<c2))

Сравнения выполняются только для контейнеров одного и того же типа. Таким образом, элементы *и* критерий сортировки должны иметь одинаковые типы; в противном случае во время компиляции возникнет ошибка. Рассмотрим пример:

```
std::set<float> c1;                          // критерий сортировки: std::less<>
std::set<float,std::greater<float> > c2;
...
if (c1 == c2) {                              // ОШИБКА: разные типы
...
}
```

Проверка, является ли один контейнер меньше другого, основана на лексикографическом сравнении (см. раздел 11.5.4). Для сравнения контейнеров разных типов (с разными критериями сортировки) следует использовать алгоритмы сравнения, описанные в разделе 11.5.4.

Специальные операции поиска

Поскольку множества и мультимножества оптимизированы для поиска элементов, они имеют специальные функции для поиска (табл. 7.36). Эти функции являются специальными версиями общих алгоритмов, имеющих те же имена. Для того чтобы достичь логарифмической, а не линейной сложности, характерной для общих алгоритмов, следует отдать предпочтение оптимизированным версиям функций поиска для множеств и мультимножеств. Например, поиск в коллекции из 1000 в среднем требует только 10 сравнений, а не 500 (см. раздел 2.2).

Таблица 7.36. Специальные функции поиска для множеств и мультимножеств

Операция	Действие
`c.count(`*val*`)`	Возвращает количество элементов, имеющих значение *val*
`c.find(`*val*`)`	Возвращает позицию первого элемента, имеющего значение *val* (или позицию `end()`, если элемент не найден)

Окончание табл. 7.36

Операция	Действие
c.lower_bound(*val*)	Возвращает первую позицию, в которую можно вставить элемент со значением *val* (первый элемент, удовлетворяющий условию >= *val*)
c.upper_bound(*val*)	Возвращает последнюю позицию, в которую можно вставить элемент со значением *val* (первый элемент, удовлетворяющий условию > *val*)
c.equal_range(*val*)	Возвращает интервал со всеми элементами, значения которых равны *val* (т.е. первую и последнюю позиции, в которые можно вставить элемент со значением *val*)

Функция-член `find()` ищет первый элемент, имеющий значение, переданное как аргумент, и возвращает ее позицию итератора. Если такой элемент не найден, функция `find()` возвращает позицию `end()` контейнера.

Функции-члены `lower_bound()` и `upper_bound()` возвращают первую и последнюю позиции соответственно, в которые можно вставить элемент с переданным значением. Иначе говоря, функция-член `lower_bound()` возвращает позицию первого элемента, который равен или больше аргумента, а функция-член `upper_bound()` возвращает позицию первого элемента, который больше аргумента. Функция-член `equal_range()` возвращает значения, возвращаемые функциями-членами `lower_bound()` и `upper_bound()`, в виде объекта класса `pair` (тип `pair` описан в разделе 5.1.1). Таким образом, функция-член `equal_range()` возвращает интервал элементов, значения которых совпадают со значениями аргумента. Если позиция, возвращаемая функцией-членом `lower_bound()`, или первое значение пары, возвращаемой функцией-членом `equal_range()`, равно позиции, возвращаемой функцией-членом `upper_bound()`, или второму значению пары, возвращаемой функцией-членом `equal_range()`, то в множестве или мультимножестве нет элементов, имеющих искомое значение. Естественно, интервал элементов множества, имеющих одинаковые значения, может содержать не больше одного элемента.

Следующий пример демонстрирует использование функций-членов `lower_bound()`, `upper_bound()` и `equal_range()`:

```
// cont/setrange1.cpp

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

int main ()
{
   set<int> c;

   c.insert(1);
   c.insert(2);
   c.insert(4);
   c.insert(5);
   c.insert(6);

   cout << "lower_bound(3): " << *c.lower_bound(3) << endl;
```

```
    cout << "upper_bound(3): " << *c.upper_bound(3) << endl;
    cout << "equal_range(3): " << *c.equal_range(3).first << " "
         << *c.equal_range(3).second << endl;

    cout << endl;
    cout << "lower_bound(5): " << *c.lower_bound(5) << endl;
    cout << "upper_bound(5): " << *c.upper_bound(5) << endl;
    cout << "equal_range(5): " << *c.equal_range(5).first << " "
         << *c.equal_range(5).second << endl;
}
```

Результат работы программы выглядит следующим образом:

```
lower_bound(3): 4
upper_bound(3): 4
equal_range(3): 4 4
lower_bound(5): 5
upper_bound(5): 6
equal_range(5): 5 6
```

Если вместо множества используется мультимножество, программа выдает такие же результаты.

Операции присваивания

Как указано в табл. 7.37, множества и мультимножества предусматривают только основные операции присваивания, характерные для всех контейнеров (см. раздел 7.1.2).

Таблица 7.37. Операции присваивания множеств и мультимножеств

Операция	Действие
c = *c2*	Присваивает все элементы контейнера *c2* контейнеру *c*
c = *rv*	Перемещает все элементы контейнера *rv* контейнеру *c* (по стандарту C++11)
c = *initlist*	Присваивает контейнеру *c* все элементы списка инициализации *initlist* (по стандарту C++11)
c1.swap(*c2*)	Обменивает данные контейнеров *c1* и *c2*
swap(*c1*,*c2*)	Обменивает данные контейнеров *c1* и *c2*

При выполнении этих операций оба контейнера должны иметь одинаковый тип. В частности, тип критерия сравнения должен быть тем же самым, хотя сам критерий сравнения может быть другим. Пример другого критерия сравнения, имеющего тот же самый тип, описан в разделе 7.7.5. Если критерии различны, они также присваиваются или обмениваются.

Функции для итераторов

Множества и мультимножества не предоставляют прямой доступ к элементам, поэтому необходимо использовать цикл `for` (см. раздел 3.1.4) или итераторы. Множества и мультимножества содержат обычные функции-члены для работы с итераторами (табл. 7.38).

Таблица 7.38. Функции для итераторов множеств и мультимножеств

Операция	Действие
`c.begin()`	Возвращает двунаправленный итератор, установленный на первый элемент
`c.end()`	Возвращает двунаправленный итератор, установленный на позицию, следующую за последним элементом
`c.cbegin()`	Возвращает константный двунаправленный итератор, установленный на первый элемент (по стандарту C++11)
`c.cend()`	Возвращает константный двунаправленный итератор, установленный на позицию, следующую за последним элементом (начиная со стандарта C++11)
`c.rbegin()`	Возвращает обратный итератор, установленный на первый элемент в обратном обходе
`c.rend()`	Возвращает обратный итератор, установленный на позицию, следующую за последним элементом в обратном обходе
`c.crbegin()`	Возвращает константный обратный итератор, установленный на первый элемент в обратном обходе (по стандарту C++11)
`c.crend()`	Возвращает константный обратный итератор, установленный на позицию, следующую за последним элементом в обратном обходе (по стандарту C++11)

Как и во всех ассоциативных контейнерах, итераторы множеств и мультимножеств являются двунаправленными (см. раздел 9.2.4). Таким образом, их нельзя использовать с алгоритмами, предназначенными для работы только с итераторами произвольного доступа.

Более важным является ограничение, которое заключается в том, что с точки зрения итератора все элементы считаются константными. Необходимо гарантировать, что при изменении значений элементов порядок их следования не будет нарушен. Однако вследствие этого ограничения к элементам множеств и мультимножеств невозможно применить ни один модифицирующий алгоритм. Например, нельзя вызвать алгоритм `remove()`, потому что он удаляет элементы путем замещения удаленных элементов следующими элементами (см. раздел 6.7.2). Для удаления элементов во множествах и мультимножествах можно использовать только члены-функции контейнера.

Вставка и удаление элементов

В табл. 7.39 перечислены операции, предусмотренные для вставки и удаления элементов множеств и мультимножеств.

Таблица 7.39. Операции вставки и удаления элементов множеств и мультимножеств

Операция	Действие
`c.insert(`*val*`)`	Вставляет копию значения *val*, возвращает позицию нового элемента и признак успешного выполнения вставки (для множеств)
`c.insert(`*pos*`,` *val*`)`	Вставляет копию значения *val* и возвращает позицию нового элемента (параметр *pos* используется как подсказка, указывающая на позицию, с которой следует начинать поиск места для вставки)

Окончание табл. 7.39

Операция	Действие
`c.insert(beg,end)`	Вставляет копии всех элементов интервала [*beg*,*end*) (не возвращая ничего)
`c.insert(initlist)`	Вставляет копию всех элементов списка инициализации *initlist* (не возвращая ничего; по стандарту C++11)
`c.emplace(args...)`	Вставляет новый элемент, инициализированный списком аргументов args, и возвращает позицию нового элемента признак успешного выполнения вставки (для множеств) (по стандарту C++11)
`c.emplace_hint(pos,args...)`	Вставляет новый элемент, инициализированный списком аргументов args, и возвращает позицию нового элемента (параметр *pos* используется как подсказка, указывающая на позицию, с которой следует начинать поиск места для вставки)
`c.erase(val)`	Удаляет все элементы, равные *val*, и возвращает количество удаленных элементов
`c.erase(pos)`	Удаляет элемент, занимающий позицию, на которую указывает итератор *pos*, и возвращает следующую позицию (до появления стандарта C++11 не возвращал ничего)
`c.erase(beg,end)`	Удаляет все элементы интервала [*beg*,*end*) и возвращает следующую позицию (до появления стандарта C++11 не возвращал ничего)
`c.clear()`	Удаляет все элементы (опустошает контейнер)

Как обычно при использовании библиотеки STL, программист должен гарантировать, что аргументы являются корректными. Итераторы должны ссылаться на корректные позиции, а начало интервала должно предшествовать его концу.

Вставка и удаление выполняются быстрее, если при работе с несколькими элементами мы используем один вызов операции для элементов, а не несколько вызовов.

Для мультимножеств стандарт C++11 гарантирует, что функции-члены `insert()`, `emplace()` и `erase()` сохраняют относительный порядок следования эквивалентных элементов, а вставляемые элементы размещаются после существующих эквивалентных значений.

Обратите внимание на то, что типы значений, возвращаемых функциями-членами для вставки элементов `insert()` и `emplace()`, отличаются друг от друга.

- **Множества** предоставляют следующий интерфейс[10]:

```
pair<iterator,bool>    insert (const value_type& val);
iterator               insert (const_iterator posHint,
                               const value_type& val);
template <typename...  Args>
  pair<iterator, bool> emplace (Args&&... args);
template <typename...  Args>
  iterator             emplace_hint (const_iterator posHint,
                                     Args&&... args);
```

- **Мультимножества** предоставляют следующий интерфейс[10]:

```
iterator               insert (const value_type& val);
iterator               insert (const_iterator posHint,
```

[10] До появления стандарта C++11 множества содержали только функцию-член `insert()`, а член `posHint` имел тип `iterator`, а не `const_iterator`.

```
                                const value_type& val);
template <typename... Args>
  iterator              emplace (Args&&... args);
template <typename... Args>
  iterator              emplace_hint (const_iterator posHint,
                                      Args&&... args);
```

Разница между типами возвращаемых значений объясняется тем, что мультимножества допускают дубликаты, а множества — нет. Таким образом, вставка элемента в множество может завершиться неудачно, если оно уже содержит элемент с таким же значением. По этой причине тип возвращаемого значения для множества представляет собой структуру `pair` (структура `pair` обсуждается в разделе 5.1.1).

1. Член `second` структуры `pair` указывает, успешно ли завершилась вставка.
2. Член `first` структуры `pair` содержит позицию вновь вставленного элемента или позицию ранее существовавшего элемента.

Во всех остальных случаях функции возвращают позицию нового или существующего элемента, если множество уже содержит элемент с таким же значением.

Следующий пример демонстрирует, как этот интерфейс используется для вставки нового элемента в множество. Этот фрагмент кода пытается вставить элемент со значением `3.3` в множество `c`:

```
std::set<double> c;
...
if (c.insert(3.3).second) {
    std::cout << "3.3 inserted" << std::endl;
}
else {
    std::cout << "3.3 already exists" << std::endl;
}
```

Код для обработки новой или старой позиции выглядит немного сложнее:

```
// вставляем новое значение и обрабатываем возврат
auto status = c.insert(value);
if (status.second) {
    std::cout << value << " inserted as element "
}
else {
    std::cout << value << " already exists as element "
}
std::cout << std::distance(c.begin(),status.first) + 1 << std::endl;
```

Результат работы этого фрагмента программы может быть таким:

```
8.9 inserted as element 4
7.7 already exists as element 3
```

В этом примере переменная `status` имеет тип

```
std::pair<std::set<float>::iterator,bool>
```

Отметим, что типы значений, возвращаемых функциями вставки с дополнительным позиционным параметром, не отличаются друг от друга. Эти функции возвращают единственный итератор как для множеств, так и для мультимножеств. Однако эти функции приводят к тем же результатам, что и функции без позиционного параметра. Разница заключается лишь в производительности. Вы можете передать позицию итератора, но она интерпретируется лишь как подсказка для осуществления оптимизации по быстродействию. Фактически, если элемент вставляется сразу после позиции, переданной как первый аргумент, сложность изменяется с логарифмической на амортизированную константную (сложность обсуждается в разделе 2.2). Тот факт, что все функции вставки с дополнительным позиционным параметром имеют один и тот же тип возвращаемого значения, позволяет создавать обобщенный код, вставляющий элементы в любые контейнеры (за исключением массивов и последовательных списков. Этот интерфейс используется обобщенными операциями вставки. Детали изложены в разделе 9.4.2.

Для удаления элемента, имеющего заданное значение, можно просто вызвать функцию-член `erase()`.

```
std::set<Elem> coll;
...
// удаляем все элементы с переданным значением
coll.erase(value);
```

Эта функция-член имеет имя, которое отличается от имени функции `remove()`, предусмотренной для списков (обсуждение функции `remove()` изложено в разделе 7.5.2). Эта функция отличается тем, что она возвращает количество удаленных элементов. Когда она применяется к множествам, она возвращает только 0 или 1.

Если мультимножество содержит дубликаты, использовать функцию `erase()` для удаления только первых экземпляров дубликатов нельзя. Вместо этого следует написать такой код:

```
std::multiset<Elem> coll;
...
// удаляем первый элемент, имеющий переданное значение
std::multiset<Elem>::iterator pos;
pos = coll.find(value);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

Для того чтобы код работал быстрее, здесь следует использовать функцию-член `find()`, а не алгоритм `find()`.

До появления стандарта C++11 функции-члены `erase()` в ассоциативных контейнерах ничего не возвращали (т.е. тип их возвращаемого значения был `void`). Причина заключалась в производительности. Поиск и возврат следующего элемента в ассоциативном массиве может оказаться затратным, потому что контейнер реализуется в виде бинарного дерева. Однако это обстоятельство значительно усложняет код, в котором элементы удаляются во время обхода (см. раздел 7.8.2).

Обратите также внимание на то, что для множеств, использующих итераторы как элементы, вызов функции-члена `erase()` может оказаться неоднозначным. По этой причине стандарт C++11 был исправлен и теперь содержит перегруженные варианты функций-членов `erase(iterator)` и `erase(const_iterator)`.

Для мультимножеств все функции-члены `insert()`, `emplace()` и `erase()` сохраняют относительный порядок следования эквивалентных элементов. В соответствии со стандартом C++11 вызов функций `insert(val)` или `emplace(args...)` гарантирует, что новый элемент будет вставлен в конец интервала эквивалентных элементов.

7.7.3. Обработка исключений

Множества и мультимножества — это контейнеры, основанные на узлах, поэтому любая неудача при попытке создать узел оставляет контейнер без изменений. Более того, поскольку деструкторы в общем случае не генерируют исключений, удаление узла не может завершиться неудачей.

Однако для операций вставки нескольких элементов необходимость сохранять их упорядоченность делает полное восстановление после генерации исключения непрактичным. Таким образом, все операции вставки единственных элементов следуют принципу "все или ничего". Иначе говоря, они или успешно выполняются, или ничего не делают. Кроме того, гарантируется, что все операции удаления нескольких элементов всегда выполняются успешно или не выполняют никаких действий, при условии, что критерий сравнения не генерирует исключений. Если операции копирования/присваивания критерия сравнения могут генерировать исключения, то то же относится и к функции-члену `swap()`.

Обработка исключений в библиотеке STL обсуждается в разделе 6.12.2.

7.7.4. Примеры использования множеств и мультимножеств

Следующая программа демонстрирует некоторые возможности множеств:

```
// cont/set1.cpp

#include <iostream>
#include <set>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
   // тип коллекции:
   // - без дубликатов
   // - элементы имеют целочисленные значения
   // - упорядочение по убыванию
   set<int,greater<int>> coll1;

   // вставляем элементы в случайном порядке, используя разные функции-члены
   coll1.insert({4,3,5,1,6,2});
   coll1.insert(5);
```

```
    // выводим на экран все элементы
    for (int elem : coll1) {
        cout << elem << ' ';
    }
    cout << endl;

    // снова вставляем 4 и обрабатываем возвращаемое значение
    auto status = coll1.insert(4);
    if (status.second) {
        cout << "4 inserted as element "
             << distance(coll1.begin(),status.first) + 1 << endl;
    }
    else {
        cout << "4 already exists" << endl;
    }

    // присваиваем элементы другому множеству, в котором элементы
    // следуют в возрастающем порядке
    set<int> coll2(coll1.cbegin(),coll1.cend());

    // выводим на экран копию, используя потоковые итераторы
    copy (coll2.cbegin(), coll2.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl;

    // удаляем все элементы до элемента со значением 3
    coll2.erase (coll2.begin(), coll2.find(3));

    // удаляем все элементы со значением 5
    int num;
    num = coll2.erase (5);
    cout << num << " element(s) removed" << endl;

    // выводим на экран все элементы
    copy (coll2.cbegin(), coll2.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

Во-первых, создается пустое множество, в которое с помощью перегруженных версий функции `insert()` вставляются несколько элементов.

```
set<int,greater<int>> coll1;

coll1.insert({4,3,5,1,6,2});
coll1.insert(5);
```

Отметим, что элемент со значением 5 вставляется дважды. Однако вторая вставка игнорируется, потому что множество не допускает дубликатов.

После вывода на экран всех элементов программа снова пытается вставить элемент 4. На этот раз она обрабатывает значение, возвращаемое функцией-членом `insert()`, как описано в разделе 7.7.2.

Код

```
set<int> coll2(coll1.cbegin(),coll1.cend());
```

создает новое множество чисел типа int, следующих в возрастающем порядке, и инициализирует их элементами старого множества. Оба контейнера имеют разные критерии сортировки, поэтому их типы отличаются, и мы не можем присваивать или сравнивать их непосредственно. Однако мы можем использовать алгоритмы, которые в общем случае способны обрабатывать разные контейнерные типы, лишь бы типы их элементов совпадали или могли преобразовываться один в другой.

Следующий оператор удаляет все элементы вплоть до элемента со значением 3.

```
coll2.erase (coll2.begin(), coll2.find(3));
```

Поскольку элемент со значением 3 расположен к конце интервала, он не удаляется.

В заключение удаляются все элементы, имеющие значение 5.

```
int num;
num = coll2.erase (5);
cout << num << " element(s) removed" << endl;
```

Результат работы программы приведен ниже.

```
6 5 4 3 2 1
4 already exists
1 2 3 4 5 6
1 element(s) removed
3 4 6
```

Для мультимножества та же самая программа (см. файл cont/multiset1.cpp) выглядит немного иначе и выдает другие результаты. Во-первых, все упоминания типа set заменяются типом multiset (заголовочный файл остается прежним).

```
multiset<int,greater<int>> coll1;
...
multiset<int> coll2(coll1.cbegin(),coll1.cend());
```

Кроме того, обработка значения, возвращаемого функцией-членом insert(), выглядит иначе. Множества не допускают дубликатов, поэтому функция-член insert() возвращает новую позицию вставленного элемента и признак того, что операция была завершена успешно.

```
auto status = coll1.insert(4);
if (status.second) {
    cout << "4 inserted as element "
         << distance(coll1.begin(),status.first) + 1 << endl;
}
else {
    cout << "4 already exists" << endl;
}
```

Для мультимножеств функция-член insert() возвращает только новую позицию (поскольку мультимножества могут содержать дубликаты, вставка может завершиться неудачно, только если возникнет исключение).

```
auto ipos = coll1.insert(4);
cout << "4 inserted as element "
     << distance(coll1.begin(),ipos) + 1 << endl;
```

Результат работы программы выглядит следующим образом:

```
6 5 4 3 2 1
4 already exists
1 2 3 4 5 6
1 element(s) removed
3 4 6
```

7.7.5. Пример задания критерия сортировки во время выполнения программы

Обычно критерий сортировки указывают как часть типа — либо в виде второго шаблонного аргумента, либо в виде критерия сортировки по умолчанию `less<>`. Однако иногда критерий сортировки необходимо задавать или изменять во время выполнения программы. В таких случаях необходим специальный тип критерия сортировки, позволяющий передавать параметры сортировки во время выполнения программы. Пример такого критерия описан в следующей программе[11]:

```
// cont/setcmp1.cpp

#include <iostream>
#include <set>
#include "print.hpp"
using namespace std;

// тип для критерия сортировки, задаваемого
// во время выполнения программы
class RuntimeCmp {
public:
   enum cmp_mode {normal, reverse};
private:
   cmp_mode mode;
public:
   // конструктор для критерия сортировки
   // - критерий по умолчанию использует значение normal
   RuntimeCmp (cmp_mode m=normal) : mode(m) {
   }
   // сравнение элементов
   // - функция-член для любого типа элементов
   template <typename T>
   bool operator() (const T& t1, const T& t2) const {
      return mode==normal ? t1<t2
                          : t2<t1;
   }
   // сравнение критериев сортировки
```

[11] Благодарю за этот пример Даниэля Крюглера.

```
    bool operator== (const RuntimeCmp& rc) const {
      return mode == rc.mode;
    }
};

// Тип множества, использующего этот критерий сортировки
typedef set<int,RuntimeCmp> IntSet;

int main()
{
    // создаем, заполняем и выводим на экран
    // множество с обычным порядком элементов
    // - используем критерий сортировки, заданный по умолчанию
    IntSet coll1 = { 4, 7, 5, 1, 6, 2, 5 };
    PRINT_ELEMENTS (coll1, "coll1: ");

    // создает критерий сортировки с обратным порядком
    RuntimeCmp reverse_order(RuntimeCmp::reverse);

    // создаем, заполняем и выводим на экран
    // множество с обратным порядком элементов
    IntSet coll2(reverse_order);
    coll2 = { 4, 7, 5, 1, 6, 2, 5 };
    PRINT_ELEMENTS (coll2, "coll2: ");

    // присваиваем элементы И критерий сортировки
    coll1 = coll2;
    coll1.insert(3);
    PRINT_ELEMENTS (coll1, "coll1: ");

    // проверка ...
    if (coll1.value_comp() == coll2.value_comp()) {
        cout << "coll1 and coll2 have the same sorting criterion"
             << endl;
    }
    else {
        cout << "coll1 and coll2 have a different sorting criterion"
             << endl;
    }
}
```

В этой программе класс `RuntimeCmp` позволяет задавать во время выполнения программы критерий сортировки любого типа. Его конструктор по умолчанию упорядочивает элементы в возрастающем порядке, используя по умолчанию значение `normal`. Кроме того, можно передать параметр `RuntimeCmp::reverse` для сортировки элементов в обратном порядке.

Результат работы программы выглядит следующим образом:

```
coll1: 1 2 4 5 6 7
coll2: 7 6 5 4 2 1
coll1: 7 6 5 4 3 2 1
coll1 and coll2 have the same sorting criterion
```

Контейнеры `coll1` и `coll2` имеют один и тот же тип, в то время как при передаче функциональных объектов `less<>` и `greater<>` в качестве критериев сортировки это не так. Кроме того, оператор присваивания выполняет присваивание элементов *и* критерия сортировки; в противном случае присваивание легко испортило бы критерий сортировки.

7.8. Отображения и мультиотображения

Отображения и мультиотображения — это контейнеры, элементами которых являются пары "ключ–значение". Эти контейнеры автоматически упорядочивают свои элементы в соответствии с определенным критерием сортировки, заданным для ключа. Разница между отображениями и мультиотображениями заключается в том, что мультиотображения допускают дубликаты, а отображения — нет (рис. 7.14).

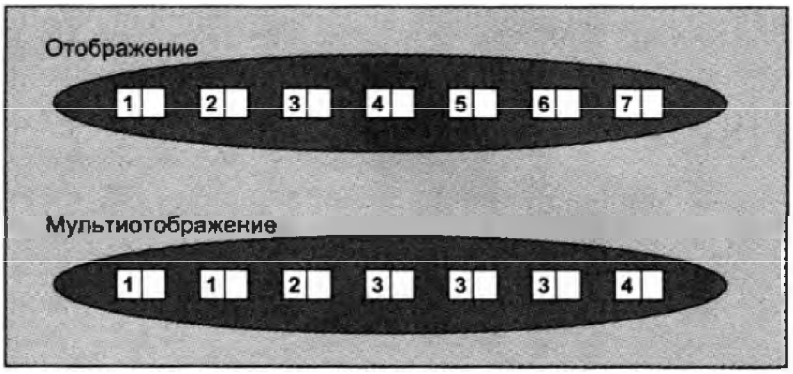


Рис. 7.14. Отображения и мультиотображения

Для использования отображений и мультиотображений в программу необходимо включить заголовочный файл `<map>`.

```
#include <map>
```

В этом файле данные типы определены как шаблонные классы в пространстве имен `std`.

```
namespace std {
   template <typename Key, typename T,
             typename Compare = less<Key>,
             typename Allocator = allocator<pair<const Key,T> > >
             class map;
   template <typename Key, typename T,
             typename Compare = less<Key>,
             typename Allocator = allocator<pair<const Key,T> > >
             class multimap;
}
```

Первый шаблонный параметр — это тип ключа элемента, а второй шаблонный параметр — этот тип значения, связанного с элементом. Элемент отображения или мультиотображения может иметь любые типы `Key` и `T`, удовлетворяющие следующим условиям.

1. Ключ и значение должны допускать копирования или перемещение.
2. Тип ключа должен быть совместимым с критерием сортировки.

Отметим, что тип элемента (`value_type`) — это пара `pair <const Key, T>`.

Необязательный третий шаблонный параметр определяет критерий сортировки. Как и в множествах, этот критерий сортировки должен определять "строгое слабое упорядочение" (см. раздел 7.7). Элементы упорядочиваются по их ключам, поэтому значения элементов не влияют на порядок их следования. Критерий сортировки используется также для проверки эквивалентности; иначе говоря, два элемента считаются равными, если ни один из их ключей не меньше другого.

Если в контейнер не передается специальный критерий сортировки, по умолчанию используется критерий `less<>`. Функциональный объект `less<>` упорядочивает элементы, сравнивая их с помощью оператора < (подробное описание функционального объекта `less` приведено в разделе 10.2.1).

В мультиотображениях порядок элементов с эквивалентными ключами случаен, но стабилен — вставки и удаления сохраняют относительный порядок эквивалентных элементов (это гарантируется стандартом C++11).

Необязательный четвертый шаблонный параметр определяет модель памяти (см. главу 19). По умолчанию используется модель памяти `allocator`, предоставленная стандартной библиотекой C++.

7.8.1. Возможности отображений и мультиотображений

Как и все стандартные ассоциативные контейнерные классы, отображения и мультиотображения обычно реализуются в виде сбалансированных бинарных деревьев (рис. 7.15). Стандарт это условие не регламентирует, но оно следует из сложности операций над отображениями и мультиотображениями. Фактически множества, мультимножества, отображения и мультиотображения используют один и тот же внутренний тип данных. Следовательно, множества и мультимножества можно рассматривать как особый вид отображений и мультиотображений соответственно, в которых значение и ключ — один и тот же объект. Таким образом, отображения и мультиотображения имеют все возможности и операции, которые имеют множества и мультимножества. Тем не менее существуют небольшие различия. Во-первых, их элементами являются пары "ключ–значение". Во-вторых, отображения можно использовать как ассоциативные массивы.

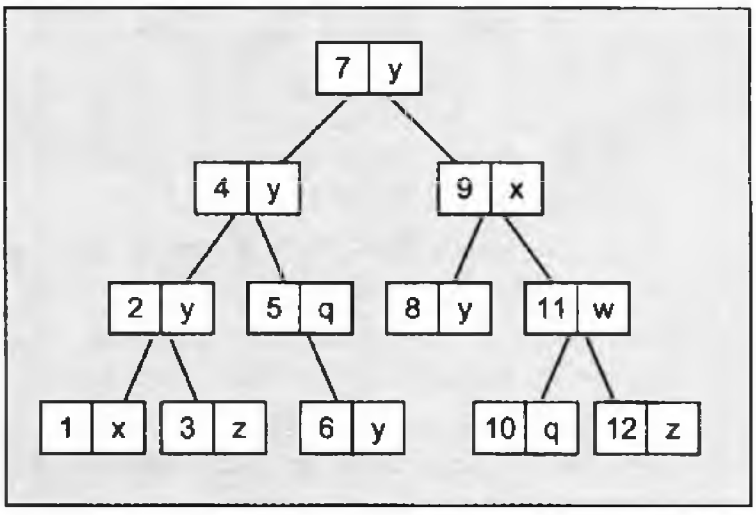


Рис. 7.15. Внутренняя структура отображений и мультиотображений

Отображения и мультиотображения автоматически упорядочивают свои элементы по их ключам. Это обеспечивает хорошее быстродействие при поиске элементов по заданному ключу. Поиск элементов по заданному значению имеет плохое быстродействие. Автоматическая сортировка накладывает важное ограничение на отображения и мультиотображения: ключ элемента *нельзя* изменить непосредственно, потому что это может нарушить правильный порядок элементов. Для модификации ключа элемента необходимо удалить элемент со старым ключом и вставить новый элемент с новым ключом и старым значением (детали описаны в разделе 7.8.2). Вследствие этого с точки зрения итератора ключ элемента является константой. Однако непосредственная модификация значения элемента остается возможной, если его тип не является константным.

7.8.2. Операции над отображениями и мультиотображениями

Создание, копирование и удаление

В табл. 7.40 перечислены конструкторы и деструкторы отображений и мультиотображений.

Таблица 7.40. Конструкторы и деструкторы отображений и мультиотображений

Операция	Действие
map *c*	Конструктор по умолчанию; создает пустое отображение/мультиотображение, не содержащее никаких элементов
map *c* (*op*)	Создает пустое отображение/мультиотображение, использующее операцию *op* в качестве критерия сортировки
map *c* (*c2*)	Копирующий конструктор; создает копию другого отображения/мультиотображения того же типа (все элементы копируются)
map *c* = *c2*	Копирующий конструктор; создает копию другого отображения/мультиотображения того же типа (все элементы копируются)
map *c* (*rv*)	Перемещающий конструктор; создает новое отображение/мультиотображение того же типа, получающее содержимое контейнера *rv* (по стандарту C++11)
map *c* = *rv*	Перемещающий конструктор; создает новое отображение/мультиотображение того же типа, получающее содержимое контейнера *rv* (по стандарту C++11)
map *c* (*beg*, *end*)	Создает отображение/мультиотображение, инициализированное элементами интервала [*beg*,*end*)
map *c* (*beg*, *end*, *op*)	Создает отображение/мультиотображение с критерием сортировки *op*, инициализированное элементами интервала [*beg*,*end*)
map *c* (*initlist*)	Создает отображение/мультиотображение, инициализированное элементами списка инициализации *initlist* (по стандарту C++11)
map *c* = *initlist*	Создает отображение/мультиотображение, инициализированное элементами списка инициализации *initlist* (по стандарту C++11)
c.~***map***()	Уничтожает все элементы и освобождает память

Окончание табл. 7.40

Отображение	Действие
Здесь слово ***map*** может означать один из следующих типов.	
`map<`*Key,Val*`>`	Отображение, по умолчанию упорядочивающее ключи с помощью критерия `less<>` (оператор <)
`map<`*Key,Val,Op*`>`	Отображение, по умолчанию упорядочивающее ключи с помощью предиката *Op*
`multimap<`*Key,Val*`>`	Мультиотображение, по умолчанию упорядочивающее ключи с помощью критерия `less<>` (оператор <)
`multimap<`*Key,Val,Op*`>`	Мультиотображение, по умолчанию упорядочивающее ключи с помощью предиката *Op*

Критерий сортировки можно задать двумя способами.

1. **Как шаблонный параметр**. Например:

```
std::map<float,std::string,std::greater<float>> coll;
```

2. В этом случае критерий сортировки является частью типа. В таком случае система типов допускает сочетание только контейнеров с одинаковыми критериями сортировки. Это обычный способ задания критерия сортировки. Точнее говоря, второй параметр — эти *тип* критерия сортировки. Конкретный критерий сортировки представляет собой функциональный объект, созданный контейнером. Для этого конструктор контейнера вызывает конструктор, предусмотренный по умолчанию для типа критерия сортировки. Пример использования пользовательского критерия сортировки приведен в разделе 10.1.1.

3. **Как параметр конструктора**. В этом случае вы можете иметь тип для нескольких критериев сортировки, допускающих разные начальные значения или состояния. Это полезно при обработке критерия сортировки во время выполнения программы, а также в ситуациях, когда критерии сортировки должны быть разными, но иметь одинаковые типы. Законченный пример приведен в разделе 7.8.6.

Если не указан никакой конкретный критерий сортировки, используется функциональный объект `less<>`, заданный по умолчанию и упорядочивающий элементы по ключу с помощью оператора <. Критерий сортировки используется также для проверки эквивалентности двух элементов в одном и том же контейнере (т.е. для поиска дубликатов). Оператор == используется только для сравнения двух контейнеров.

Определение типа позволяет избежать надоедливого повторения в коде.

```
typedef std::map<std::string,float,std::greater<std::string>>
        StringFloatMap;
...
StringFloatMap coll;
```

Конструктор с инициализацией интервалом можно использовать для инициализации контейнера элементами из контейнеров, имеющих другие типы, массивов или стандартного потока (см. раздел 7.1.2). Однако элементами контейнера являются пары "ключ–значение", поэтому необходимо гарантировать, что элементы из интервала-источника имеют тип `pair<key, value>` или могут быть преобразованы в него.

Немодицифирующие операции и специальные операции поиска

Отображения и мультиотображения предусматривают обычные немодифицирующие операции, запрашивающие размер контейнера и выполняющие сравнения (табл. 7.41).

Таблица 7.41. Немодифицирующие операции над отображениями и мультиотображениями

Операция	Действие
`c.key_comp()`	Возвращает критерий сравнения
`c.value_comp()`	Возвращает критерий сравнения для значений в целом (объект, сравнивающий ключи в паре «ключ–значение»)
`c.empty()`	Возвращает результат проверки того, что контейнер пуст (эквивалент выражения `size()==0`, но может работать быстрее)
`c.size()`	Возвращает текущее количество элементов
`c.max_size()`	Возвращает максимально возможное количество элементов
c1 == *c2*	Возвращает результат проверки того, что контейнер *c1* равен контейнеру *c2* (применяя к элементам операцию ==)
c1 != *c2*	Возвращает результат проверки того, что контейнер *c1* не равен контейнеру *c2* (эквивалент ! (*c1*==*c2*))
c1 < *c2*	Возвращает результат проверки того, что контейнер *c1* меньше контейнера *c2*
c1 > *c2*	Возвращает результат проверки того, что контейнер *c1* больше контейнера *c2* (эквивалент *c2*<*c1*)
c1 <= *c2*	Возвращает результат проверки того, что контейнер *c1* меньше или равен контейнеру *c2* (эквивалент ! (*c2*<*c1*))
c1 >= *c2*	Возвращает результат проверки того, что контейнер *c1* больше или равен контейнеру *c2* (эквивалент ! (*c1*<*c2*))

Сравнения выполняются только для контейнеров одного и того же типа. Таким образом, ключ, значение и критерий сортировки должны иметь одинаковые типы; в противном случае во время компиляции возникнет ошибка.

```
std::map<float,std::string> c1;          // критерий сортировки: less<>
std::map<float,std::string,std::greater<float> > c2;
...
if (c1 == c2) {                          // ОШИБКА: разные типы
...
}
```

Проверка, является ли один контейнер меньше другого, основана на лексикографическом сравнении (см. раздел 11.5.4). Для сравнения контейнеров разных типов (с разными критериями сортировки) следует использовать алгоритмы сравнения, описанные в разделе 11.5.4.

Специальные операции поиска

Как и множества и мультимножества, отображения и мультиотображения имеют специальные функции-члены для поиска, обеспечивающие высокую производительность благодаря внутренней древовидной структуре этих контейнеров (табл. 7.42).

Таблица 7.42. Специальные функции поиска для множеств и мультимножеств

Операция	Действие
c.count(*val*)	Возвращает количество элементов, имеющих ключ *val*
c.find(*val*)	Возвращает позицию первого элемента, имеющего ключ *val* (или позицию end(), если таковой не найден)
c.lower_bound(*val*)	Возвращает первую позицию, в которую можно вставить элемент с ключом *val* (первый элемент, удовлетворяющий условию >= *val*)
c.upper_bound(*val*)	Возвращает последнюю позицию, в которую можно вставить элемент с ключом *val* (первый элемент, удовлетворяющий условию > *val*)
c.equal_range(*val*)	Возвращает интервал со всеми элементами, значения которых равны *val* (т.е. первую и последнюю позиции, в которые можно вставить элемент с ключом *val*)

Функция-член find() ищет первый элемент, имеющий заданный ключ, и возвращает его позицию итератора. Если такой элемент не найден, функция find() возвращает позицию end() контейнера. С помощью функции-члена find() невозможно найти элемент с заданным значением. Вместо нее в таком случае следует использовать универсальный алгоритм, такой как find_if(), или явный цикл. Рассмотрим пример простого цикла, выполняющего некие операции над каждым элементом, имеющим заданное значение:

```
std::multimap<std::string,float> coll;
...
// делаем что-то со всеми элементами, имеющими заданное значение
std::multimap<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
     if (pos->second == value) {
         do_something(pos);
     }
}
```

При удалении элементов с помощью такого цикла следует быть осторожным. Вы можете спилить сук, на котором сидите! Более подробно этот вопрос рассмотрен в разделе 7.8.2.

Использование алгоритма find_if() для поиска элемента, имеющего заданное значение, еще сложнее, чем цикл, потому что необходимо передать ему функциональный объект, сравнивающий значение элемента с указанным значением. Соответствующий пример приведен в разделе 7.8.5.

Функции-члены lower_bound(), upper_bound() и equal_range() работают точно так же, как и их аналоги для множеств (см. раздел 7.7.2), за исключением того, что элементами теперь являются пары "ключ–значение".

Операции присваивания

Как указано в табл. 7.43, отображения и мультиотображения предусматривают только основные операции присваивания, которые есть во всех контейнерах (см. раздел 7.1.2).

Таблица 7.43. Операции присваивания для множеств и мультимножеств

Операция	Действие
c = *c2*	Присваивает все элементы контейнера *c2* контейнеру *c*
c = *rv*	Перемещает все элементы контейнера *rv* контейнеру *c* (по стандарту C++11)
c = *initlist*	Присваивает контейнеру *c* все элементы списка инициализации *initlist* (по стандарту C++11)
c1.swap(*c2*)	Обменивает данные контейнеров *c1* и *c2*
swap(*c1*,*c2*)	Обменивает данные контейнеров *c1* и *c2*

При выполнении этих операций оба контейнера должны иметь одинаковый тип. В частности, одинаковым должен быть тип критерия сравнения, хотя сам критерий сравнения может быть другим. Пример разных критериев сравнения, имеющих один и тот же тип, описан в разделе 7.8.6. Если критерий отличается, контейнеры можно присваивать или обменивать.

Функции для итераторов и доступа к элементам

Множества и мультимножества не предоставляют прямой доступ к элементам, поэтому необходимо использовать диапазонный цикл `for` (см. раздел 3.1.4) или итераторы. Исключение составляют лишь функция-член `at()` и оператор индексирования, предоставляющие прямой доступ к элементам (см. раздел 7.8.3).

В табл. 7.44 перечислены обычные функции-члены для работы с итераторами отображений и мультиотображений.

Таблица 7.44. Операции над итераторами множеств и мультимножеств

Операция	Действие
c.begin()	Возвращает двунаправленный итератор, установленный на первый элемент
c.end()	Возвращает двунаправленный итератор, установленный на позицию, следующую за последним элементом
c.cbegin()	Возвращает константный двунаправленный итератор, установленный на первый элемент (по стандарту C++11)
c.cend()	Возвращает константный двунаправленный итератор, установленный на позицию, следующую за последним элементом (начиная со стандарта C++11)
c.rbegin()	Возвращает обратный итератор, установленный на первый элемент в обратном обходе
c.rend()	Возвращает обратный итератор, установленный на позицию, следующую за последним элементом в обратном обходе
c.crbegin()	Возвращает константный обратный итератор, установленный на первый элемент в обратном обходе (по стандарту C++11)
c.crend()	Возвращает константный обратный итератор, установленный на позицию, следующую за последним элементом в обратном обходе (по стандарту C++11)

Как и во всех ассоциативных контейнерах, итераторы множеств и мультимножеств являются двунаправленными (см. раздел 9.2.4). Таким образом, их нельзя использовать с алгоритмами, предназначенными для работы только с итераторами произвольного доступа.

Более важным является ограничение, которое заключается в том, что с точки зрения итератора ключи всех элементов считаются константными. Иначе говоря, элементы имеют тип `pair<const Key, T>`. Это необходимо для того, чтобы гарантировать, что при изменении значений элементов порядок их следования не будет нарушен. Однако вследствие этого ограничения к элементам множеств и мультимножеств невозможно применить ни один модифицирующий алгоритм. Например, нельзя вызвать алгоритм `remove()`, потому что он "удаляет" элементы путем замещения "удаленных" элементов следующими за ними элементами (см. раздел 6.7.2). Для удаления элементов из отображений и мультиотображений можно использовать только члены-функции контейнера.

Следующий пример демонстрирует доступ к элементам с помощью цикла `for` по интервалу:

```
std::map<std::string,float> coll;
...
for (auto& elem : coll) {
     std::cout << "key: " << elem.first << "\t"
               << "value: " << elem.second << std::endl;
}
```

В этом цикле переменная `elem` становится ссылкой на текущий элемент контейнера `coll`. Таким образом, переменная `elem` имеет тип `pair<const std::string,float>`. Выражение `elem.first` возвращает ключ текущего элемента, а выражение `elem.second` возвращает его значение.

Соответствующий код, использующий итератор, который надо было использовать до появления стандарта C++11, выглядит следующим образом:

```
std::map<std::string,float> coll;
...
std::map<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
     std::cout << "key: " << pos->first << "\t"
               << "value: " << pos->second << std::endl;
}
```

Здесь итератор `pos` перемещается по последовательности пар, состоящих из объектов типов `const string` и `float`, а для доступа к ключу и значению текущего элемента необходимо использовать оператор `->`.[12]

Попытка изменить значение ключа приводит к ошибке.

```
elem.first = "hello"; // ОШИБКА во время компиляции
pos->first = "hello"; // ОШИБКА во время компиляции
```

Однако изменение значения элемента не представляет проблемы, поскольку переменная `elem` объявлена как неконстантная ссылка, а тип значения не является константным.

```
elem.second = 13.5; // OK
pos->second = 13.5; // OK
```

[12] Выражение `pos->first` является сокращением выражения `(*pos).first`.

При использовании алгоритмов и лямбда-функций для манипулирования элементами отображений необходимо явно объявлять тип элемента.

```
std::map<std::string,float> coll;
...
// добавляем 10 к значению каждого элемента:
std::for_each (coll.begin(), coll.end(),
               [] (std::pair<const std::string,float>& elem) {
                   elem.second += 10;
               });
```

Для объявления типа элемента вместо использования класса

```
std::pair<const std::string,float>
```

можно использовать тип

```
std::map<std::string,float>::value_type
```

или

```
decltype(coll)::value_type
```

Законченный пример см. в разделе 7.8.5.

Для изменения ключа элемента существует только одна возможность: необходимо заменить старый элемент новым с тем же значением. Обобщенная функция, выполняющая эту процедуру, выглядит следующим образом:

```
// cont/newkey.hpp

namespace MyLib {
   template <typename Cont>
   inline
   bool replace_key (Cont& c,
                     const typename Cont::key_type& old_key,
                     const typename Cont::key_type& new_key)
   {
      typename Cont::iterator pos;
      pos = c.find(old_key);
      if (pos != c.end()) {
          // вставляем новый элемент со значением старого элемента
          c.insert(typename Cont::value_type(new_key,
                                             pos->second));
          // удаляем старый элемент
          c.erase(pos);
          return true;
      }
      else {
          // ключ не найден
          return false;
      }
   }
}
```

Функции-члены `insert()` и `erase()` обсуждаются в следующем разделе.

Для использования этой обобщенной функции достаточно просто передать контейнер, а также старый и новый ключи. Например:

```
std::map<std::string,float> coll;
...
MyLib::replace_key(coll,"old key","new key");
```

Точно так же эта функция работает с мультиотображениями (заменяя *первый* совпадающий ключ).

Отметим, что отображение предоставляет более удобный способ для модификации ключа элемента. Вместо вызова функции-члена `replace_key()` можно просто написать следующий код:

```
// вставляет новый элемент со значением старого элемента
coll["new_key"] = coll["old_key"];
// удаляем старый элемент
coll.erase("old_key");
```

Детали использования операции индексирования отображений приведены в разделе 7.8.3.

Вставка и удаление элементов

В табл. 7.45 перечислены операции для вставки и удаления элементов отображений и мультиотображений. К ним относятся все замечания, сделанные в разделе 7.7.2 для множеств и мультимножеств. В частности, типы значений, возвращаемых этими операциями, отличаются друг от друга точно так же, как в множествах и мультимножествах. Однако следует подчеркнуть, что элементами отображений и мультиотображений являются пары "ключ–значение". Таким образом, использование этих функций немного сложнее.

Таблица 7.45. Операции вставки и удаления элементов отображений и мультиотображений

Операция	Действие
c.insert(*val*)	Вставляет копию значения *val*, возвращает позицию нового элемента и признак успешного выполнения вставки (для отображений)
c.insert(*pos*,*val*)	Вставляет копию значения *val* и возвращает позицию нового элемента (параметр *pos* используется как подсказка, указывающая на позицию, с которой следует начинать поиск места для вставки)
c.insert(*beg*,*end*)	Вставляет копии всех элементов интервала [*beg*,*end*) (не возвращает ничего)
c.insert(*initlist*)	Вставляет копию всех элементов списка инициализации *initlist* (не возвращает ничего; по стандарту C++11)
c.emplace(*args*...)	Вставляет новый элемент, инициализированный списком аргументов *args*, возвращает позицию нового элемента и признак успешного выполнения вставки (для отображений) (по стандарту C++11)

Окончание табл. 7.45

Операция	Действие
`c.emplace_hint(pos,args...)`	Вставляет новый элемент, инициализированный списком аргументов *args*, и возвращает позицию нового элемента (параметр *pos* используется как подсказка, указывающая на позицию, с которой следует начинать поиск места для вставки)
`c.erase(val)`	Удаляет все элементы, равные *val*, и возвращает количество удаленных элементов
`c.erase(pos)`	Удаляет элемент, занимающий позицию итератора *pos*, и возвращает следующую позицию (до появления стандарта C++11 не возвращал ничего)
`c.erase(beg,end)`	Удаляет все элементы интервала [*beg*,*end*) и возвращает следующую позицию (до появления стандарта C++11 не возвращал ничего)
`c.clear()`	Удаляет все элементы (опустошает контейнер)

Для мультиотображений стандарт C++11 гарантирует, что функции-члены `insert()`, `emplace()` и `erase()` сохраняют относительный порядок эквивалентных элементов, а вставленные элементы размещаются в конце существующих эквивалентных значений.

При вставке пары "ключ–значение" необходимо иметь в виду, что в множествах и мультимножествах ключ считается константным. Необходимо либо указать корректный тип, либо предусмотреть неявное или явное преобразование типа.

По стандарту C++11 для вставки элементов удобнее всего использовать функцию-член `emplace()` или передать их функции-члену `insert()` в виде списка инициализации, в котором первым элементом является ключ, а вторым — значение.

```
std::map<std::string,float> coll;
...
coll.emplace("jim",17.7);   // если для инициализации ключа/значения
                            // необходимо больше аргументов,
                            // см. ниже
coll.insert({"otto",22.3});
```

Существуют три других альтернативных способа передачи значения в отображение и мультиотображение.

1. **Использование типа `value_type`**. Для того чтобы избежать неявного преобразования типа, можно передать корректный тип явно с помощью типа `value_type`, который определяется контейнерным типом. Например:

   ```
   std::map<std::string,float> coll;
   ...
   coll.insert(std::map<std::string,float>::value_type("otto", 22.3));
   ```

   или

   ```
   coll.insert(decltype(coll)::value_type("otto",22.3));
   ```

2. **Использование типа `pair<>`**. Другим способом является явное использование типа `pair<>`. Например:

```
std::map<std::string,float> coll;
...
// использование неявного преобразования:
coll.insert(std::pair<std::string,float>("otto",22.3));
// использование явного преобразования:
coll.insert(std::pair<const std::string,float>("otto",22.3));
```

В первом операторе, вызывающем функцию-член `insert()`, тип указан не совсем правильно, поэтому он преобразовывается в тип реального элемента. Для этого функция-член `insert()` определяется как шаблонная функция (см. раздел 3.2).

3. **Использование функции `make_pair()`**. Вероятно, наиболее удобным способом до появления стандарта C++11 было использование функции `make_pair()`, создающей пару, состоящую из двух значений, передаваемых как аргументы (см. раздел 5.1.1).

```
std::map<std::string,float> coll;
...
coll.insert(std::make_pair("otto",22.3));
```

Как и прежде, необходимые преобразования типов выполняются шаблонной функцией-членом `insert()`. Рассмотрим простой пример вставки элемента в отображение, в котором осуществляется проверка ее успешности.

```
std::map<std::string,float> coll;
...
if (coll.insert(std::make_pair("otto",22.3)).second) {
    std::cout << "OK, could insert otto/22.3" << std::endl;
}
else {
    std::cout << "OOPS, could not insert otto/22.3 "
              << "(key otto already exists)" << std::endl;
}
```

Обсуждение типов значений, возвращаемых разными вариантами функции-члена `insert()`, а также дополнительные примеры, которые можно применять и к отображениям, см. в разделе 7.7.2. Вновь отметим, что отображения предусматривают удобные операцию `[]` и функцию-член `at()` для вставки (и задания) элементов с помощью операции индексирования (см. раздел 7.8.3).

При вставке нового элемента с помощью функции `emplace()`, а также при инициализации ключей и/или элементов несколькими значениями, необходимо передавать два списка аргументов: один для ключа, другой — для значения. Наиболее удобный способ описан ниже.

```
std::map<std::string,std::complex<float>> m;

m.emplace(std::piecewise_construct,  // передаем в качестве аргументов
                                     // кортежи элементов
          std::make_tuple("hello"),  // элементы для ключа
          std::make_tuple(3.4,7.8)); // элементы для значения
```

Подробности процедуры создания пар из отдельных компонентов см. в разделе 5.1.1.

Для удаления элемента, имеющего конкретное значение, можно просто вызвать функцию-член `erase()`.

```
std::map<std::string,float> coll;
...
// удаление всех элементов с заданным ключом
coll.erase(key);
```

Эта версия функции-члена `erase()` возвращает количество удаленных элементов. При работе с отображениями функция `erase()` может возвращать только 0 или 1.

Если мультиотображение содержит дубликаты и требуется удалить первый из них, не следует вызывать функцию-член `erase()`. Вместо этого можно воспользоваться следующим кодом:

```
std::multimap<std::string,float> coll;
...
// удаляем первый элемент с заданным ключом
auto pos = coll.find(key);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

Здесь необходимо использовать функцию-член `find()`, а не алгоритм `find()`, потому что функция работает быстрее (см пример в разделе 7.3.2). Однако для удаления элементов, имеющих заданное значение, а не ключ, функцию-член `find()` применять нельзя. Обсуждение этого вопроса см. в разделе 7.8.2.

При удалении элементов следует проявлять осторожность, чтобы не спилить сук, на котором сидите. Существует большая опасность того, что вы удалите элемент, на который ссылается ваш итератор. Например:

```
std::map<std::string,float> coll;
...
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    if (pos->second == value) {
        coll.erase(pos); // RUNTIME ERROR !!!
    }
}
```

Применение функции-члена `erase()` к элементу, на который ссылается итератор `pos`, делает его некорректным по отношению к контейнеру `coll`. Поэтому, если вы попытаетесь использовать итератор `pos` после удаления его элемента без повторной инициализации, ситуация кардинально изменится. Фактически простое выполнение операции `++pos` приводит к непредсказуемым последствиям.

Стандарт C++11 упрощает решение, так как функция-член `erase()` всегда возвращает значение следующего элемента:

```
std::map<std::string,float> coll;
...
for (auto pos = coll.begin(); pos != coll.end(); ) {
    if (pos->second == value) {
        pos = coll.erase(pos); // возможно только начиная со стандарта C++11
    }
```

```
    else {
        ++pos;
    }
}
```

К сожалению, до появления стандарта C++11 проектное решение не предусматривало возвращения следующей позиции, потому что, если оно не было необходимым, это приводило к излишним затратам времени. Однако в результате работа программистов усложнялась и становилась подверженной ошибкам. Кроме того, обработка этих ошибок оказывалась еще дороже в смысле затрат времени. Рассмотрим пример правильного удаления элементов, в котором итераторы используются в стиле, принятом до появления стандарта C++11:

```
typedef std::map<std::string,float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
// удаляем все элементы, имеющие заданное значение
for (pos = coll.begin(); pos != coll.end(); ) {
    if (pos->second == value) {
        coll.erase(pos++);
    }
    else {
        ++pos;
    }
}
```

Отметим, что оператор `pos++` увеличивает итератор `pos` так, что он ссылается на следующий элемент, но возвращает копию оригинального значения. Таким образом, итератор `pos` не ссылается на элемент, удаленный функцией `erase()`.

Кроме того, в множествах, использующих итераторы в качестве элементов, вызов функции-члена `erase()` теперь может оказаться неоднозначным. По этой причине стандарт C++11 был исправлен, чтобы обеспечить перегрузку версий `erase(iterator)` и `erase(const_iterator)`.

Для мультиотображений все функции-члены `insert()`, `emplace()` и `erase()` сохраняют относительный порядок эквивалентных элементов. Стандарт C++11 гарантирует, что при вызове функций `insert(`*val*`)` или `emplace(`*args*`...)` новый элемент будет вставлен в конец интервала, содержащего эквивалентные элементы.

7.8.3. Использование отображений как ассоциативных массивов

Ассоциативные контейнеры обычно не обеспечивают прямой доступ к элементам. Вместо этого следует использовать итераторы. Однако для отображений, как и для неупорядоченных отображений (см. раздел 7.9), существует исключение из этого правила. Неконстантные отображения имеют операцию индексирования для прямого доступа к элементам.

Кроме того, стандарт C++11 предусматривает соответствующую функцию-член `at()` для константных и неконстантных отображений (табл. 7.46).

Таблица 7.46. Прямой доступ к элементам отображений

Операция	Действие
c[*key*]	Вставляет элемент с ключом *key*, если его еще не было, и возвращает ссылку на значение элемента с ключом *key* (только для неконстантных отображений)
c.at(*key*)	Возвращает ссылку на значение элемента с ключом *key* (по стандарту C++11)

Функция-член `at()` возвращает значение элемента с заданным ключом и генерирует исключение типа `out_of_range`, если такого элемента нет.

В операторе `[ ]` индекс также является ключом, используемым для идентификации элемента. Это значит, что в операторе `[ ]` индекс может иметь любой тип, а не только целочисленный. Такой интерфейс называется *ассоциативным массивом*.

Тип индекса в операторе `[ ]` для ассоциативных массивов — не единственное отличие от обычных массивов. Кроме того, индекс не бывает неправильным. Если ключом является индекс, для которого не существует ни одного элемента, в отображение автоматически вставляется новый элемент. Значение нового элемента инициализируется конструктором по умолчанию для соответствующего типа. Таким образом, для использования этой возможности нельзя использовать тип значения, не имеющего конструктора, заданного по умолчанию. Отметим, что все элементарные типы имеют конструкторы по умолчанию, инициализирующие их значения нулями (см. раздел 3.2.1).

Это поведение ассоциативного массива имеет как преимущества, так и недостатки.

- Преимущество заключается в более удобном интерфейсе для вставки новых элементов в отображение. Например:

```
std::map<std::string,float> coll; // пустая коллекция
// вставляем "otto"/7.7 как пару ключ/значение
// - сначала вставляется пара "otto"/float()
// - затем выполняется присваивание 7.7
coll["otto"] = 7.7;
```

- Код

```
coll["otto"] = 7.7;
```

выполняется следующим образом.

1. Вычисляется выражение `coll["otto"]`.
   - Если элемент с ключом `"otto"` существует, то это выражение возвращает значение элемента по ссылке.
   - Если, как в нашем примере, элемента с ключом `"otto"` нет, выражение автоматически вставляет новый элемент с ключом `"otto"` и значением, заданным конструктором типа значения по умолчанию. Затем это выражение возвращает ссылку на новое значение нового элемента.
2. Присваивание значения `7.7`.
   - Вторая часть кода присваивает число `7.7` значению нового или существующего элемента.

После этого отображение содержит элемент с ключом `"otto"` и значение `7.7`.

- Недостаток заключается в том, что новый элемент можно вставить случайно или по ошибке. Например, следующий оператор делает нечто, чего вы не ожидали:

```
std::cout << coll["ottto"];
```

- Он вставляет новый элемент с ключом `"ottto"` и выводит на экран его значение, равное по умолчанию `0`. Однако вместо этого он должен был сгенерировать исключение, сообщив, что ключ `"otto"` написан неправильно.
- Отметим также, что такой способ вставки элементов медленнее, чем обычный способ вставки элементов в отображения, описанный в разделе 7.8.2. Причина заключается в том, что новое значение сначала инициализируется значением, заданным по умолчанию для его типа, и лишь затем перезаписывается правильным значением.

Несколько примеров приведено в разделах 6.2.4 и 7.8.5.

7.8.4. Обработка исключений

Отображения и мультиотображения по отношению к исключениям ничем не отличаются от множеств и мультимножеств. Особенности обработки исключений этими контейнерами описаны в разделе 7.7.3.

7.8.5. Примеры использования отображений и мультиотображений

Использование алгоритмов и лямбда-функций для отображений и мультиотображений

В разделе 6.2.3 приведен пример, посвященный неупорядоченным мультиотображениям, который можно использовать и для иллюстрации обычного (упорядоченного) отображения или мультиотображения. Рассмотрим соответствующий пример, иллюстрирующий отображение. Эта программа также демонстрирует использование алгоритмов и лямбда-функций вместо цикла `for` по интервалу:

```
// cont/map1.cpp

#include <map>
#include <string>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    map<string,double> coll { { "tim", 9.9 },
                              { "struppi", 11.77 }
                            } ;

   // возводим в квадрат каждый элемент:
```

```
    for_each (coll.begin(), coll.end(),
              [] (pair<const string,double>& elem) {
                  elem.second *= elem.second;
              });

    // выводим на экран каждый элемент:
    for_each (coll.begin(), coll.end(),
              [] (const map<string,double>::value_type& elem) {
                    cout << elem.first << ": " << elem.second << endl;
              });
}
```

Как видим, для отображения алгоритм `for_each()` вызывается дважды: для возведения в квадрат каждого элемента и для вывода каждого элемента. При первом вызове тип элемента объявляется явно; при втором выводе используется тип `value_type`. При первом вызове элемент передается по ссылке, чтобы можно было модифицировать его значение; при втором вызове используется константная ссылка, не допускающая нежелательное создание копий.

Программа выводит на экран следующие строки:

```
struppi: 138.533
tim: 98.01
```

Использование отображения как ассоциативного массива

Следующий пример демонстрирует использование отображения как ассоциативного массива. Отображение представляет собой диаграмму биржевого курса. Элементами отображения являются пары, в которых ключом является название акции, а значением является ее цена.

```
// cont/map2.cpp

#include <map>
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // создаем отображение / ассоциативный массив
    // - ключами являются строки
    // - значениями являются числа с плавающей точкой
    typedef map<string,float> StringFloatMap;

    StringFloatMap stocks; // создаем пустой контейнер

    // вставляем несколько элементов
    stocks["BASF"] = 369.50;
    stocks["VW"] = 413.50;
    stocks["Daimler"] = 819.00;
    stocks["BMW"] = 834.00;
```

```
    stocks["Siemens"] = 842.20;

    // выводим на экран все элементы
    StringFloatMap::iterator pos;
    cout << left; // left-adjust values
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
        cout << "stock: " << setw(12) << pos->first
             << "price: " << pos->second << endl;
    }
    cout << endl;

    // биржевой бум (все цены удваиваются)
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
        pos->second *= 2;
    }

    // выводим на экран все элементы
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
        cout << "stock: " << setw(12) << pos->first
             << "price: " << pos->second << endl;
    }
    cout << endl;

    // переименовываем ключ "VW" в "Volkswagen"
    // - задаем только изменяемый элемент
    stocks["Volkswagen"] = stocks["VW"];
    stocks.erase("VW");

    // выводим на экран все элементы
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
        cout << "stock: " << setw(12) << pos->first
             << "price: " << pos->second << endl;
    }
}
```

Программа выводит на экран следующие строки:

```
stock: BASF        price: 369.5
stock: BMW         price: 834
stock: Daimler     price: 819
stock: Siemens     price: 842.2
stock: VW          price: 413.5

stock: BASF        price: 739
stock: BMW         price: 1668
stock: Daimler     price: 1638
stock: Siemens     price: 1684.4
stock: VW          price: 827

stock: BASF        price: 739
stock: BMW         price: 1668
stock: Daimler     price: 1638
stock: Siemens     price: 1684.4
stock: Volkswagen  price: 827
```

Использование мультиотображения в качестве словаря

Следующий пример демонстрирует использование мультиотображения в качестве словаря:

```
// cont/multimap1.cpp

#include <map>
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   // создаем мультиотображение в виде словаря строка/строка
   multimap<string,string> dict;

   // вставляем элементы в случайном порядке
   dict.insert ( { {"day","Tag"}, {"strange","fremd"},
                   {"car","Auto"}, {"smart","elegant"},
                   {"trait","Merkmal"}, {"strange","seltsam"},
                   {"smart","raffiniert"}, {"smart","klug"},
                   {"clever","raffiniert"} } );

      // выводим на экран все элементы
      cout.setf (ios::left, ios::adjustfield);
      cout << ' ' << setw(10) << "english "
           << "german " << endl;
      cout << setfill('-') << setw(20) << ""
           << setfill(' ') << endl;
      for ( const auto& elem : dict ) {
            cout << ' ' << setw(10) << elem.first
                 << elem.second << endl;
   }
   cout << endl;

   // выводим на экран все значения с ключом "smart"
   string word("smart");
   cout << word << ": " << endl;
   for (auto pos = dict.lower_bound(word);
        pos != dict.upper_bound(word);
        ++pos) {
             cout << " " << pos->second << endl;
   }

   // выводим на экран все ключи для значения "raffiniert"
   word = ("raffiniert");
   cout << word << ": " << endl;
   for (const auto& elem : dict) {
        if (elem.second == word) {
            cout << " " << elem.first << endl;
        }
   }
}
```

Эта программа выводит на экран строки

```
english    german
-------------------
car        Auto
clever     raffiniert
day        Tag
smart      elegant
smart      raffiniert
smart      klug
strange    fremd
strange    seltsam
trait      Merkmal

smart:
    elegant
    raffiniert
    klug
raffiniert:
    clever
    smart
```

Соответствующий пример использования неупорядоченного мультиотображения в качестве словаря приведен в разделе 7.9.7.

Поиск элементов с определенными значениями

Нижеприведенный пример демонстрирует использование алгоритма `find_if()` для поиска элемента с определенным значением (в противоположность поиску элемента с определенным значением).

```
// cont/mapfind1.cpp

#include <map>
#include <iostream>
#include <algorithm>
#include <utility>
using namespace std;

int main()
{
   // отображение, содержащее в качестве ключа и    // значения числа с плавающей
точкой
   // - инициализация ключей и значений автоматически
   // преобразовывает их в тип float
   map<float,float> coll = { {1,7}, {2,4}, {3,2}, {4,3},
                             {5,6}, {6,1}, {7,3} };

   // поиск элемента с ключом 3.0 (логарифмическая сложность)
   auto posKey = coll.find(3.0);
   if (posKey != coll.end()) {
       cout << "key 3.0 found ("
            << posKey->first << ":"
```

```
            << posKey->second << ")" << endl;
    }

    // поиск элемента со значением 3.0 (линейная сложность)
    auto posVal = find_if(coll.begin(),coll.end(),
                          [] (const pair<float,float>& elem) {
                              return elem.second == 3.0;
                          });
    if (posVal != coll.end()) {
        cout << "value 3.0 found ("
             << posVal->first << ":"
             << posVal->second << ")" << endl;
    }
}
```

Эта программа выводит на экран следующие строки:

```
key 3.0 found (3:2)
value 3.0 found (4:3)
```

7.8.6. Пример с отображениями, строками и критериями сортировки, задаваемыми во время выполнения программы

Этот пример предназначен для опытных программистов, а не новичков. Он демонстрирует как мощь библиотеки STL, так и проблемы, связанные с ней. В частности, данный пример демонстрирует:

- как использовать отображения, включая интерфейс ассоциативных массивов;
- как писать и использовать функциональные объекты;
- как определять критерий сортировки во время выполнения программы;
- как сравнивать строки без учета регистра.

```
// cont/mapcmp1.cpp

#include <iostream>
#include <iomanip>
#include <map>
#include <string>
#include <algorithm>
#include <cctype>
using namespace std;

// функциональный объект для сравнения строк
// - позволяет задавать критерий сортировки во время выполнения программы
// - обеспечивает сравнение без учета регистра символов
class RuntimeStringCmp {
   public:
      // константы для критерия сравнения
      enum cmp_mode {normal, nocase};
```

```
  private:
    // фактический режим сравнения
    const cmp_mode mode;

  // вспомогательная функция для сравнения
  // символов без учета регистра
  static bool nocase_compare (char c1, char c2) {
    return toupper(c1) < toupper(c2);
  }
public:
  // конструктор: инициализирует критерий сравнения
  RuntimeStringCmp (cmp_mode m=normal) : mode(m) {
}

  // сравнение
  bool operator() (const string& s1, const string& s2) const {
    if (mode == normal) {
        return s1<s2;
    }
    else {
        return lexicographical_compare (s1.begin(), s1.end(),
                                        s2.begin(), s2.end(),
                                        nocase_compare);
    }
  }
};

// контейнерный тип:
// - отображение, содержащее
//    - строки в качестве ключей
//    - строки в качестве значений
//    - специальный тип функционального объекта для сравнения
typedef map<string,string,RuntimeStringCmp> StringStringMap;

// функция, заполняющая контейнер и выводящая на экран его элементы
void fillAndPrint(StringStringMap& coll);

int main()
{
  // создаем контейнер с критерием сравнения, заданным по умолчанию
  StringStringMap coll1;
  fillAndPrint(coll1);

  // создаем объект для сравнения без учета регистра
  RuntimeStringCmp ignorecase(RuntimeStringCmp::nocase);

  // создаем контейнер с критерием сравнения без учета регистра
  StringStringMap coll2(ignorecase);
  fillAndPrint(coll2);
}

void fillAndPrint(StringStringMap& coll)
{
```

```
    // вставляем элементы в случайном порядке
    coll["Deutschland"] = "Germany";
    coll["deutsch"] = "German";
    coll["Haken"] = "snag";
    coll["arbeiten"] = "work";
    coll["Hund"] = "dog";
    coll["gehen"] = "go";
    coll["Unternehmen"] = "enterprise";
    coll["unternehmen"] = "undertake";
    coll["gehen"] = "walk";
    coll["Bestatter"] = "undertaker";

    // вывод элементов на экран
    cout.setf(ios::left, ios::adjustfield);
    for (const auto& elem : coll) {
    cout << setw(15) << elem.first << " "
         << elem.second << endl;
    }
    cout << endl;
}
```

В этой программе функция `main()` создает два контейнера и вызывает для них функцию `fillAndPrint()`, заполняющую эти контейнеры одними и теми же элементами и выводящую на экран их содержимое. Однако эти контейнеры имеют два разных критерия сортировки.

1. Контейнер `coll1` использует функциональный объект типа `RuntimeStringCmp`, сравнивающий элементы с помощью оператора <.
2. Контейнер `coll2` использует функциональный объект типа `RuntimeStringCmp`, инициализированный значением `nocase` класса `RuntimeStringCmp`. Значение `nocase` заставляет этот функциональный объект сортировать строки без учета регистра.

Эта программа выводит на экран следующие строки:

```
Bestatter       undertaker
Deutschland     Germany
Haken           snag
Hund            dog
Unternehmen     enterprise
arbeiten        work
deutsch         German
gehen           walk
unternehmen     undertake

arbeiten        work
Bestatter       undertaker
deutsch         German
Deutschland     Germany
gehen           walk
Haken           snag
Hund            dog
Unternehmen     undertake
```

Первый блок содержит элементы первого контейнера, сравнивающего элементы с помощью оператора <. Сначала выводятся ключи в верхнем регистре, за которыми следуют ключи в нижнем регистре.

Второй блок содержит все элементы без учета регистра, поэтому порядок изменяется. Отметим однако, что второй блок содержит на одну строку меньше, потому что слово в верхнем регистре "Unternehmen" с точки зрения регистра совпадает со словом "unternehmen", набранным в нижнем регистре[13], а мы используем отображения, которые не допускают дубликаты. К сожалению, в результате возникла путаница, потому что немецкий ключ, инициализированный переводом слова "enterprise", получил значение "undertake". Таким образом, в этом случае, вероятно, следовало бы использовать мультиотображение. Это имеет смысл еще и потому, что для словарей естественным контейнером являются мультиотображения.

7.9. Неупорядоченные контейнеры

Хеш-таблица, одна из важных структур данных для создания коллекций, не входила в первую версию стандартной библиотеки C++. Она также не была частью исходного варианта библиотеки STL, и комитет решил, что предложение включить ее в стандарт C++98 поступило слишком поздно. (На определенном этапе программист должен прекратить изобретать новые возможности и сосредоточиться на технических деталях. В противном случае работа никогда не закончится.) Однако начиная с документа TR1 контейнеры с характеристиками хеш-таблиц стали частью стандарта.

Тем не менее, даже с появлением документа TR1, в сообществе C++ существовало несколько реализаций хеш-таблиц. Библиотеки обычно содержали четыре варианта хеш-таблиц: `hash_set`, `hash_multiset`, `hash_map` и `hash_multimap`. Однако реализации этих хеш-таблиц немного отличались друг от друга.

В документе TR1 была введена консолидированная группа контейнеров, основанных на хеш-таблицах. Функциональные возможности стандартных классов сочетали свойства существующих реализаций и не совпадали полностью ни с одной из них. По этой причине, для того чтобы избежать совпадения имен, были выбраны другие имена классов. Было решено поставить перед именами всех существующих ассоциативных контейнеров префикс `unordered_`. Это также демонстрировало наиболее важное различие между обычными и новыми ассоциативными контейнерами: в реализациях, основанных на хеш-таблицах, элементы не имеют определенного порядка. Подробности проектных решений, касающихся неупорядоченных контейнеров, описаны в работе [*N1456:HashTable*].

Строго говоря, в стандартной библиотеке C++ неупорядоченные контейнеры называются "неупорядоченными ассоциативными контейнерами". Однако я буду писать просто "неупорядоченные контейнеры". Под "ассоциативными контейнерами" я подразумеваю "старые" ассоциативные контейнеры, предусмотренные стандартом C++98 и реализованные в виде бинарных деревьев (множество, мультимножество, отображение и мультиотображение).

Теоретически неупорядоченные контейнеры содержат все вставленные элементы в произвольном порядке (рис. 7.16). Иначе говоря, контейнер можно рассматривать как мешок: в него можно класть элементы, но когда вы открываете мешок для того, чтобы сделать что-то с его элементами, вы достаете их в случайном порядке. Итак, в противоположность (мульти)множествам и (мульти)отображениям в неупорядоченных контейнерах

[13] В немецком языке все существительные начинаются с прописной буквы, а все глаголы — со строчной.

нет критерия сортировки; в противоположность последовательным контейнерам в неупорядоченных контейнерах невозможно разместить элемент в определенной позиции.

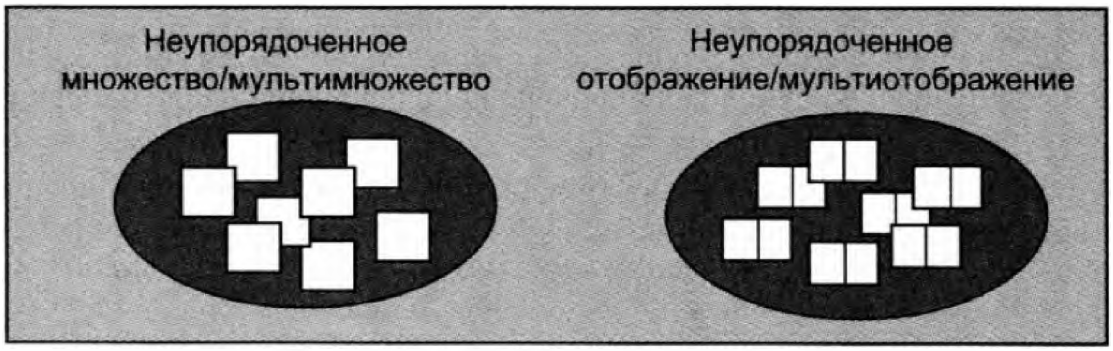


Рис. 7.16. Неупорядоченные контейнеры

Как и ассоциативные контейнеры, отдельные классы отличаются друг от друга следующими особенностями.

- Неупорядоченные множества и мультимножества хранят отдельные значения конкретного типа, а неупорядоченные отображения и мультиотображения хранят пары "ключ–значение", в которых ключ используется для хранения и поиска конкретного элемента, включая поиск связанного с ним значения.
- Неупорядоченные множества и отображения не допускают дубликаты, а неупорядоченные мультимножества и мультиотображения допускают.

Для использования неупорядоченных множеств и мультимножеств в программу необходимо включить заголовочный файл `<unordered_set>`, а для использования неупорядоченных отображений и мультиотображений в программу необходимо включить заголовочный файл `<unordered_map>`.

```
#include <unordered_set>
#include <unordered_map>
```

В этих файлах контейнерные типы определяются как шаблонные классы в пространстве имен `std`.

```
namespace std {
   template <typename T,
             typename Hash = hash<Key>,
             typename EqPred = equal_to<Key>,
             typename Allocator = allocator<T> >
             class unordered_set;

template <typename T,
          typename Hash = hash<T>,
          typename EqPred = equal_to<T>,
          typename Allocator = allocator<T> >
          class unordered_multiset;

template <typename Key, typename T,
          typename Hash = hash<T>,
```

```
        typename EqPred = equal_to<T>,
        typename Allocator = allocator<pair<const Key, T> > >
        class unordered_map;

template <typename Key, typename T,
        typename Hash = hash<T>,
        typename EqPred = equal_to<T>,
        typename Allocator = allocator<pair<const Key, T> > >
        class unordered_multimap;
}
```

Элементы неупорядоченного множества или мультимножества могут иметь любой тип T, допускающий сравнение.

Первым шаблонным параметром неупорядоченного отображения и мультиотображения является тип ключа элемента, а вторым шаблонным параметром — тип значения, связанного с элементом. Элементы неупорядоченного отображения или неупорядоченного мультиотображения могут иметь любые типы Key и T, удовлетворяющие следующим требованиям.

1. Ключ и значение должны допускать копирование или перемещение.
2. Ключ должен быть совместимым с критерием эквивалентности.

Отметим, что типом элемента (value_type) является тип pair<const Key, T>.

Необязательный второй/третий шаблонный параметр определяет хеш-функцию. Если не указана конкретная хеш-функция, используется хеш-функция hash<>, заданная по умолчанию и определенная как функциональный объект в заголовочном файле <functional> для всех целочисленных типов, всех типов чисел с плавающей точкой, указателей, строк и некоторых специальных типов[14]. Для всех других типов значений следует передавать собственную хеш-функцию, как показано в разделах 7.9.2 и 7.9.7.

Необязательный третий/четвертый шаблонный параметр определяет критерий эквивалентности: предикат, используемый для поиска элементов. Он должен возвращать признак того, что два значения являются эквивалентными. Если специальный критерий сравнения не передается, используется критерий equal_to<>, сравнивающий элементы друг с другом с помощью оператора == (подробное описание класса equal_to<> приведено в разделе 10.2.1).

Необязательный четвертый/пятый шаблонный параметр определяет модель памяти (см. раздел 19). По умолчанию используется модель памяти allocator, предусмотренная стандартной библиотекой C++.

7.9.1. Возможности неупорядоченных контейнеров

Все стандартные неупорядоченные контейнерные классы реализованы как хеш-таблицы, которые тем не менее оставляют много возможностей для совершенствования. Как обычно, стандартная библиотека C++ не определяет все детали реализаций, допускающие все возможные варианты. В то же время некоторые функциональные возможности неупорядоченных контейнеров основаны на следующих предположениях (см. [*N1456:HashTable*]).

[14] error_code, thread::id, bitset<> и vector<bool>

- Хеш-таблицы используют метод цепочек, в котором хеш-код ассоциирован со связанным списком. (Этот метод также называется "открытым хешированием" или "закрытой адресацией". Его не следует путать с "открытой адресацией" и "закрытым хешированием".)
- Выбор вида связанного списка (одно- или двухсвязного) предоставляется программисту, осуществляющему реализацию. По этой причине стандарт гарантирует лишь, что итераторы должны быть "по крайней мере" однонаправленными.
- Возможны различные стратегии повторного хеширования (рехеширования).
  - В рамках традиционного подхода в результате отдельной операции вставки или удаления время от времени происходит полная реорганизация внутренних данных.
  - В рамках инкрементного хеширования происходит постепенное изменение количества сегментов или слотов, особенно полезное в системах реального времени, в которых цена одноразового увеличения хеш-таблицы может оказаться слишком высокой.
- Неупорядоченные контейнеры допускают обе стратегии и не гарантируют, что между ними не возникнут конфликты.

На рис. 7.17 показана типичная внутренняя структура неупорядоченного множества или мультимножества в соответствии с минимальными гарантиями, предоставляемыми стандартной библиотекой C++. Для сохранения каждого значения хеш-функция отображает его в сегмент (слот) хеш-таблицы. Каждый слот управляет односвязным списком, содержащим все элементы, для которых хеш-функция вычисляет одно и то же значение.

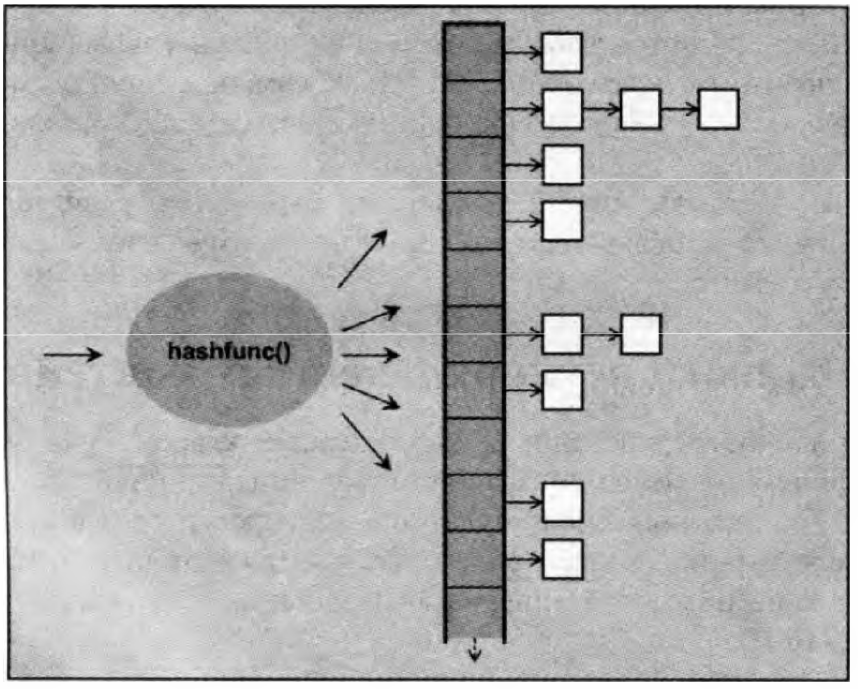


Рис. 7.17. Внутренняя структура неупорядоченных множеств и мультимножеств

На рис. 7.18 показана типичная внутренняя структура неупорядоченного отображения и мультиотображения в соответствии с минимальными гарантиями, предоставляемыми стандартной библиотекой C++. Для сохранения каждого элемента, представляющего собой пару "ключ–значение", хеш-функция отображает его в сегмент (слот) хеш-таблицы. Каждый слот управляет односвязным списком, содержащим все элементы, для которых хеш-функция вычисляет одно и то же значение.

Основное преимущество хеш-таблиц заключается в их невероятной эффективности во время выполнения программы. Если стратегия хеширования выбрана и реализована правильно, гарантируется амортизированное константное время выполнения вставок, удалений и поиска элемента ("амортизированное", потому что возможное эпизодическое рехеширование может оказаться операцией с линейной сложностью).

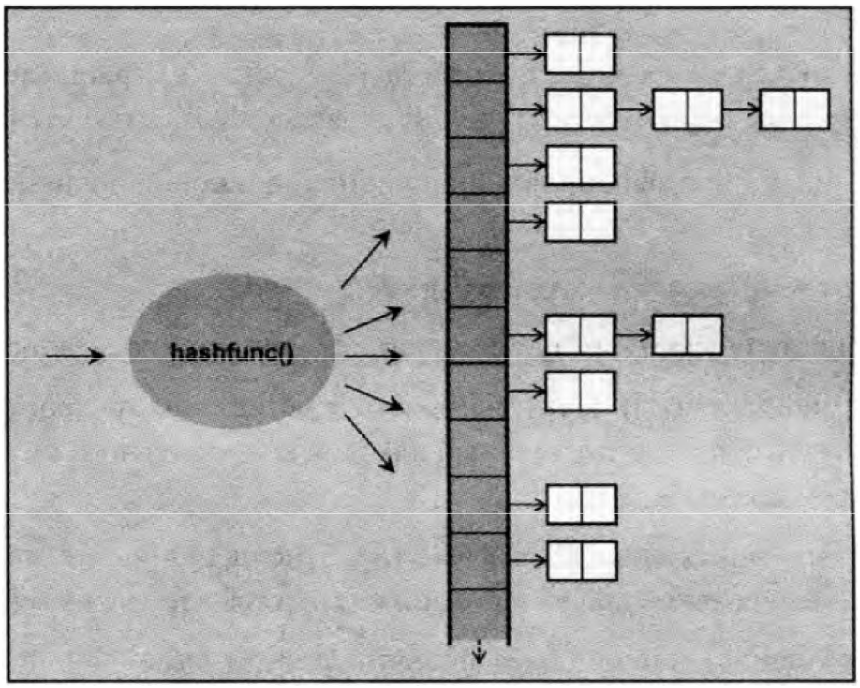


Рис. 7.18. Внутренняя структура неупорядоченных отображения и мультиотображений

Ожидаемое поведение практически всех операций над неупорядоченными контейнерами, включая копирующее конструирование и присваивание, вставку и удаление элемента, а также проверку эквивалентности, зависит от качества хеш-функции. Если хеш-функция вычисляет одинаковые значения для разных элементов, что происходит, когда неупорядоченный контейнер, допускающий дубликаты, заполнен эквивалентными значениями или ключами, любая операция над хеш-таблицей имеет низкую производительность. Этот недостаток присущ не столько самой структуре данных, сколько ее применению непросвещенными пользователями.

Неупорядоченные контейнеры имеют также недостатки по сравнению с обычными ассоциативными контейнерами.

- Неупорядоченные контейнеры не предусматривают операторов <, >, <= и >= для упорядочивания повторяющихся экземпляров в этих контейнерах. В то же время они предусматривают операторы == и != (по стандарту C++11).
- Функции-члены `lower_bound()` и `upper_bound()` не предусмотрены.

- Поскольку гарантированно итераторы могут быть только однонаправленными, обратные итераторы, включая итераторы, возвращаемые функциями-членами `rbegin()`, `rend()`, `crbegin()` и `crend()`, не поддерживаются, и невозможно применить алгоритмы, требующие двунаправленных итераторов (по крайней мере, они не являются переносимыми).

Поскольку (ключ) значение элемента определяет его позицию — в данном случае сегмент, — его *нельзя* модифицировать непосредственно. Следовательно, как и в большинстве ассоциативных контейнеров, для модификации значения элемента необходимо удалить его и вставить новый элемент с новым значением. Интерфейс отражает эту особенность.

- Неупорядоченные контейнеры не предусматривают операции для прямого доступа к элементам.
- Непрямой доступ к элементам посредством итераторов ограничен тем, что с точки зрения итератора (ключ) значение элемента является константным.

Программист может модифицировать параметры, влияющие на эффективность хеш-таблицы.

- Можно задать минимальное количество сегментов.
- Можно (а иногда нужно) предусматривать собственную хеш-функцию.
- Можно (а иногда нужно) предусматривать собственный критерий проверки эквивалентности: предикат, используемый для поиска правильного элемента среди всех записей в списках сегментов.
- Можно задать максимальный коэффициент заполнения, позволяющий выполнять повторное хеширование при увеличении количества элементов контейнера.
- Повторное хеширование можно выполнять принудительно.

На следующие факторы повлиять невозможно.

- Коэффициент роста, управляющий автоматическим хешированием, используется для уменьшения или увеличения списка сегментов.
- Коэффициент минимального заполнения, используемый для осуществления вынужденного повторного хеширования, когда количество элементов в контейнере уменьшается.

Отметим, что повторное хеширование возможно только после вызова функций-членов `insert()`, `rehash()`, `reserve()` или `clear()`. Это является следствием гарантии, что функция-член `erase()` никогда не делает некорректными итераторы и указатели на другие элементы. Таким образом, при удалении сотен элементов размер сегмента не изменится. Однако если после этого вставить всего один элемент, размер сегмента может уменьшиться.

Отметим также, что в контейнерах, содержащих эквивалентные ключи, — неупорядоченных мультимножествах и мультиотображениях — элементы с эквивалентными ключами являются смежными при обходе элементов контейнера. Повторное хеширование и другие операции, изменяющие порядок следования элементов, сохраняют порядок следования эквивалентных ключей.

7.9.2. Создание неупорядоченных контейнеров и управление ими

Хеш-таблицы представляют собой довольно сложные структуры данных. По этой причине существует масса возможностей для определения или выяснения особенностей их функционирования.

Создание, копирование и удаление

В табл. 7.47 перечислены конструкторы и деструкторы неупорядоченных ассоциативных контейнеров. В табл. 7.48 перечислены типы ***Unord***, которые можно использовать вместе с указанными конструкторами и деструкторами.

Таблица 7.47. Конструкторы и деструкторы неупорядоченных контейнеров

Операция	Действие
Unord *c*	Конструктор по умолчанию; создает пустой неупорядоченный контейнер, не содержащий никаких элементов
Unord *c* (*bnum*)	Создает пустой неупорядоченный контейнер, использующий по крайней мере *bnum* сегментов
Unord *c* (*bnum*, *hf*)	Создает пустой неупорядоченный контейнер, использующий по крайней мере *bnum* сегментов и хеш-функцию *hf*
Unord *c* (*bnum*, *hf*, *cmp*)	Создает пустой неупорядоченный контейнер, использующий по крайней мере *bnum* сегментов, хеш-функцию *hf* и предикат *cmp* для идентификации одинаковых значений
Unord *c* (*c2*)	Копирующий конструктор; создает другой неупорядоченный контейнер того же типа (все элементы копируются)
Unord *c* = *c2*	Копирующий конструктор; создает копию другого неупорядоченного контейнера того же типа (все элементы копируются)
Unord *c* (*rv*)	Перемещающий конструктор; создает неупорядоченный контейнер, получающий содержимое контейнера *rv* (по стандарту C++11)
Unord *c* = *rv*	Перемещающий конструктор; создает неупорядоченный контейнер, принимающий содержимое контейнера *rv* (по стандарту C++11)
Unord *c* (*beg*, *end*)	Создает неупорядоченный контейнер, инициализированный элементами интервала [*beg*,*end*)
Unord *c* (*beg*, *end*, *bnum*)	Создает неупорядоченный контейнер, инициализированный элементами интервала [*beg*,*end*) и использующий по крайней мере *bnum* сегментов
Unord *c* (*beg*, *end*, *bnum*, *hf*)	Создает неупорядоченный контейнер, инициализированный элементами интервала [*beg*,*end*) и использующий по крайней мере *bnum* сегментов и хеш-функцию *hf*

Окончание табл. 7.47

Операция	Действие
Unord *c* (*beg*, *end*, *bnum*, *hf*, *cmp*)	Создает неупорядоченный контейнер, инициализированный элементами интервала [*beg*,*end*) и использующий по крайней мере *bnum* сегментов, хеш-функцию *hf* и предикат *cmp* для идентификации одинаковых значений
Unord *c* (*initlist*)	Создает неупорядоченный контейнер, инициализированный элементами списка инициализации *initlist*
Unord *c* = *initlist*	Создает неупорядоченный контейнер, инициализированный элементами списка инициализации *initlist*
c.~***Unord***()	Уничтожает все элементы и освобождает память

Таблица 7.48. Возможные типы *Unord* неупорядоченных массивов

Unord	Действие
`unordered_set<`*Elem*`>`	Неупорядоченное множество, которое по умолчанию использует хеш-функцию `hash<>` и критерий сравнения `equal_to<>` (оператор ==)
`unordered_set<`*Elem*, *Hash*`>`	Неупорядоченное множество, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения `equal_to<>` (оператор ==)
`unordered_set<`*Elem*, *Hash*, *Cmp*`>`	Неупорядоченное множество, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения *Cmp*
`unordered_multiset<`*Elem*`>`	Неупорядоченное мультимножество, которое по умолчанию использует хеш-функцию `hash<>` и критерий сравнения `equal_to<>` (оператор ==)
`unordered_multiset<`*Elem*, *Hash*`>`	Неупорядоченное мультимножество, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения `equal_to<>` (оператор ==)
`unordered_multiset<`*Elem*, *Hash*, *Cmp*`>`	Неупорядоченное мультимножество, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения *Cmp*
`unordered_map<`*Key*, *T*`>`	Неупорядоченное мультиотображение, которое по умолчанию использует хеш-функцию `hash<>` и критерий сравнения `equal_to<>` (оператор ==)
`unordered_map<`*Key*, *T*, *Hash*`>`	Неупорядоченное мультиотображение, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения `equal_to<>` (оператор ==)
`unordered_map<`*Key*, *T*, *Hash*, *Cmp*`>`	Неупорядоченное отображение, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения *Cmp*
`unordered_multimap<`*Key*, *T*`>`	Неупорядоченное мультиотображение, которое по умолчанию использует хеш-функцию `hash<>` и критерий сравнения `equal_to<>` (оператор ==)

Окончание табл. 7.48

Unord	Действие
`unordered_multimap<`*Key*`,` *T*`,` *Hash*`>`	Неупорядоченное мультиотображение, которое по умолчанию использует хеш-функцию `hash<>` и критерий сравнения `equal_to<>` (оператор ==)
`unordered_multimap<`*Key*`,` *T*`,` *Hash*`,` *Cmp*`>`	Неупорядоченное мультиотображение, которое по умолчанию использует хеш-функцию *Hash* и критерий сравнения *Cmp*

При создании контейнера существует много возможностей для передачи аргументов. С одной стороны, можно передать значения как исходные элементы:

- существующий контейнер того же типа (копирующий конструктор);
- все элементы интервала [*begin*,*end*);
- все элементы списка инициализации.

С другой стороны, можно передать аргументы, влияющие на функционирование неупорядоченного контейнера:

- хеш-функция (либо как шаблонный аргумент, либо как аргумент конструктора);
- критерий эквивалентности (либо как шаблонный аргумент, либо как аргумент конструктора);
- начальное количество сегментов (как аргумент конструктора).

Отметим, что коэффициент максимального заполнения нельзя задать как часть типа или как аргумент конструктора, хотя этот параметр часто хочется иметь возможность указать изначально. Для задания коэффициента максимального заполнения следует вызвать функцию-член после создания объекта (см. табл. 7.49).

```
std::unordered_set<std::string> coll;
coll.max_load_factor(0.7);
```

Аргумент функции-члена `max_load_factor()` должен иметь тип `float`. В принципе значение между `0.7` и `0.8` обеспечивает разумный компромисс между быстродействием и объемом используемой памяти. Отметим, что по умолчанию коэффициент максимального заполнения равен `1.0`, т.е. обычно коллизии возникают до того, как будет выполнено повторное хеширование. По этой причине, если главным является быстродействие, всегда следует явно задавать коэффициент максимального заполнения.

Структурные операции

Неупорядоченные контейнеры предусматривают также операции для получения информации о внутренней структуре и воздействия на нее. Эти операции перечислены в табл. 7.49.

Таблица 7.49. Структурные операции над неупорядоченными контейнерами

Операция	Действие
c.hash_function()	Возвращает хеш-функцию
c.key_eq()	Возвращает предикат эквивалентности
c.bucket_count()	Возвращает текущее количество сегментов
c.max_bucket_count()	Возвращает максимально возможное количество сегментов
c.load_factor()	Возвращает текущий коэффициент заполнения
c.max_load_factor()	Возвращает текущий коэффициент максимального заполнения
c.max_load_factor(*val*)	Задает коэффициент максимального заполнения равным *val*
c.rehash(*bnum*)	Повторно хеширует контейнер, так, чтобы его размер сегмента был не меньше *bnum*
c.reserve(*num*)	Повторно хеширует контейнер, чтобы он мог содержать не меньше *num* элементов (по стандарту C++11)

Помимо функции-члена max_load_factor(), к важным функциям-членам относятся rehash() и reserve(). Они обеспечивают немного иной интерфейс повторного хеширования (т.е. процедуры изменения количества сегментов). В исходном варианте, в соответствии с документом TR1, была предусмотрена только функция-член rehash(), представляющая собой запрос на создание хеш-таблицы, размер сегмента которой был не меньше указанного в аргументе. Проблема заключалась в том, что при использовании такого интерфейса приходилось учитывать коэффициент максимального заполнения. Если коэффициент максимального заполнения равен 0.7 и вы хотели хранить 100 элементов, то надо было разделить 100 на 0.7 и вычислить размер, не требующий повторного хеширования, если в контейнер не было вставлено больше 100 элементов. Иначе говоря, чтобы избежать хеширования до того момента, пока в контейнере не окажется 100 элементов, функции-члену rehash() следовало передать число 143. В функции-члене reserve() это вычисление производится автоматически, поэтому можно просто передать количество элементов, для которого следует подготовить хеш-таблицу.

```
coll.rehash(100);  // подготовка для 100/max_load_factor() элементов
coll.reserve(100); // подготовка для 100 элементов (по стандарту C++11)
```

С помощью функции-члена bucket_count() можно запросить, какое количество сегментов содержит неупорядоченный контейнер. Это значение можно использовать в функциях-членах, обеспечивающих работу с сегментами, контролирующими внутреннее состояние неупорядоченного контейнера. Детали и пример проверки внутренней структуры неупорядоченного контейнера с интерфейсом сегментов приведен в разделах 7.9.4 и 7.9.7.

Создание собственной хеш-функции

Все хеш-таблицы используют хеш-функцию, отображающую значения элементов, помещаемых в контейнер, в определенный сегмент. Цель состоит в том, чтобы два равных значения приводили к одному и тому же индексу сегмента, а разные значения в идеале приводили к разным индексам. Для любого интервала передаваемых значений хеш-функция должна обеспечивать хорошее распределение хеш-значений.

Хеш-функция должна быть функцией или функциональным объектом, который получает аргумент, имеющий тип значения элемента, и возвращает значение, имеющее тип `std::size_t`. Таким образом, текущее количество сегментов во внимание не принимается. Отображение возвращаемого значения в интервал корректных индексов сегментов происходит в контейнере автоматически. Следовательно, наша цель — создать функцию, равномерно распределяющую разные значения элементов в интервале [0,`size_t`).

Вот пример предоставления пользовательской хеш-функции:

```
#include <functional>

class Customer {
...
};

class CustomerHash
{
   public:
      std::size_t operator() (const Customer& c) const {
           return ...
   }
};

std::unordered_set<Customer,CustomerHash> custset;
```

где `CustomerHash` — это функциональный объект, определяющий хеш-функцию для класса `Customer`.

Вместо того чтобы задавать функциональный объект как часть описания типа контейнера, хеш-функцию можно передать как аргумент конструктора. Отметим, однако, что при этом следует правильно указать шаблонный тип хеш-функции.

```
std::size_t customer_hash_func (const Customer& c)
{
   return ...
};

std::unordered_set<Customer,std::size_t(*)(const Customer&)>
      custset(20,customer_hash_func);
```

Здесь функция `customer_hash_func()` передается как второй аргумент конструктора, а ее тип — "указатель на функцию, получающую аргумент типа `Customer` и возвращающую значение типа `std::size_t`", — передается как второй шаблонный аргумент.

Если специальная хеш-функция не задается, то используется хеш-функция `hash<>`, заданная по умолчанию как функциональный объект и определенная в заголовке `<functional>` для "обычных" типов: всех целочисленных типов, всех чисел с плавающей точкой, указателей, строк и некоторых специальных типов[15]. Для всех других типов следует задать собственную хеш-функцию.

Написать хорошую хеш-функцию сложнее, чем кажется. Чаще всего в качестве собственных хеш-функций используют стандартные хеш-функции. Наивный подход предлагает просто суммировать все хеш-значения, атрибуты которых относятся к хеш-функциям. Например:

[15] `error_code`, `thread::id`, `bitset<>` и `vector<bool>`

```
class CustomerHash
{
   public:
      std::size_t operator() (const Customer& c) const {
           return std::hash<std::string>()(c.fname) +
                  std::hash<std::string>()(c.lname) +
                  std::hash<long>()(c.no);
   }
};
```

Здесь возвращаемое хеш-значение представляет собой сумму хеш-значений для атрибутов `fname`, `lname` и `no` класса `Customer`. Если встроенные хеш-функции для типов этих атрибутов при заданных значениях работают хорошо, то результат будет равен сумме трех значений в интервале [0,`std::size_t`). По общепринятым правилам переполнения результаты должны быть правильно распределены.

И все же эксперты считают, что это плохая хеш-функция, а создать хорошую хеш-функцию не так легко, как должно быть.

Лучший подход заключается в использовании хеш-функции из библиотеки Boost (см. [*Boost*]) и более удобного интерфейса.

```
// cont/hashval.hpp

#include <functional>

// из библиотеки boost (functional/hash):
// см. http://www.boost.org/doc/libs/1_35_0/doc/html/hash/combine.html
template <typename T>
inline void hash_combine (std::size_t& seed, const T& val)
{
   seed ^= std::hash<T>()(val) + 0x9e3779b9 + (seed<<6) + (seed>>2);
}

// вспомогательные обобщенные функции для создания хеш-значения
// по начальному значению
template <typename T>
inline void hash_val (std::size_t& seed, const T& val)
{
   hash_combine(seed,val);
}
template <typename T, typename... Types>
inline void hash_val (std::size_t& seed,
                      const T& val, const Types&... args)
{
   hash_combine(seed,val);
   hash_val(seed,args...);
}

// вспомогательная обобщенная функция для создания хеш-значений
// по списку неоднородных аргументов
template <typename... Types>
inline std::size_t hash_val (const Types&... args)
{
   std::size_t seed = 0;
```

```
   hash_val (seed, args...);
   return seed;
}
```

Удобная функция, реализованная с помощью шаблонов с переменным количеством аргументов (см. раздел 3.1.9), позволяет вызывать функцию `hash_val()` с произвольным количеством аргументов любого типа для вычисления хеш-значения по всем этим элементам. Например:

```
class CustomerHash
{
   public:
      std::size_t operator() (const Customer& c) const {
         return hash_val(c.fname,c.lname,c.no);
   }
};
```

По сути дела здесь вызывается функция `hash_combine()`, которая считается хорошим кандидатом на роль обобщенной хеш-функции (см. [*HoadZobel:HashCombine*]).

Если входные значения для хеш-функции имеют специальные ограничения, программист может написать свою хеш-функцию. В любом случае для проверки работы собственной хеш-функции можно использовать интерфейс сегментов (см. раздел 7.9.7).

Некоторые завершенные примеры, касающиеся создания собственных хеш-функций, приведены в разделе 7.9.7. Кроме того, для этого можно использовать лямбда-функции (см. раздел 7.9.7).

Создание собственного критерия эквивалентности

Третий/четвертый шаблонный параметр типа неупорядоченного контейнера — это критерий эквивалентности, т.е. предикат, который используется для поиска равных значений в одном и том же сегменте. По умолчанию в качестве этого предиката используется функциональный объект `equal_to<>`, сравнивающий элементы с помощью оператора ==. По этой причине самым удобным подходом к созданию корректного критерия эквивалентности является определение оператора == для пользовательского типа, если он не задан заранее в качестве члена класса или глобальной функции. Рассмотрим пример:

```
class Customer {
   ...
};

bool operator == (const Customer& c1, const Customer& c2) {
   ...
}

std::unordered_multiset<Customer,CustomerHash> custmset;
std::unordered_map<Customer,String,CustomerHash> custmap;
```

Впрочем, можно написать свой собственный критерий эквивалентности, как показано в следующем примере:

```
#include <functional>
class Customer {
```

```
   ...
};

class CustomerEqual
{
   public:
   bool operator() (const Customer& c1, const Customer& c2) const {
      return ...
   }
};

   std::unordered_set<Customer,CustomerHash,CustomerEqual> custset;
   std::unordered_multimap<Customer,String,
                           CustomerHash,CustomerEqual> custmmap;
```

Здесь для типа `Customer` определен функциональный объект, в котором необходимо реализовать оператор `operator()`, сравнивающий два элемента (или ключа для отображения) и возвращающий булево значение, являющееся признаком их равенства или неравенства.

Законченный пример, посвященный созданию пользовательского критерия сравнения (и хеш-функции), приведен в разделе 7.9.7. Снова напомним, что для определения критерия эквивалентности можно использовать лямбда-функции (см. раздел 7.9.7).

Если значения считаются равными в соответствии с текущим критерием эквивалентности, им должны соответствовать одинаковые хеш-значения, вычисляемые текущей хеш-функцией. По этой причине неупорядоченный контейнер, имеющий пользовательский предикат эквивалентности, должен также иметь пользовательскую хеш-функцию.

7.9.3. Другие операции над неупорядоченными контейнерами

Остальные операции над неупорядоченными контейнерами примерно совпадают с операциями над ассоциативными контейнерами.

Немодифицирующие операции

Неупорядоченные контейнеры предусматривают немодифицирующие операции, перечисленные в табл. 7.50.

Таблица 7.50. Немодифицирующие операции над неупорядоченными контейнерами

Операция	Действие
c`.empty()`	Возвращает результат проверки того, что контейнер пуст (эквивалент `size()==0`, но может работать быстрее)
c`.size()`	Возвращает текущее количество элементов
c`.max_size()`	Возвращает максимально возможное количество элементов
c1 == *c2*	Возвращает результат проверки того, что контейнер *c1* равен контейнеру *c2*
c1 != *c2*	Возвращает результат проверки того, что контейнер *c1* не равен контейнеру *c2* (эквивалент ! (*c1*==*c2*))

Снова отметим, что для сравнений неупорядоченных контейнеров предусмотрены только операции == и !=.[16] В худшем случае они могут иметь квадратичную сложность.

Специальные операции поиска

Неупорядоченные контейнеры оптимизированы для быстрого поиска элементов с заданным значением (неупорядоченные множества и мультимножества) и или ключом (неупорядоченные отображения и мультиотображения). Для того чтобы извлечь пользу из этого обстоятельства, они имеют специальные функции для поиска (табл. 7.51). Эти функции являются специальными версиями общих алгоритмов, имеющих те же имена. При работе с неупорядоченными контейнерами всегда следует использовать оптимизированные версии, обеспечивающие константную сложность в противоположность универсальным алгоритмам, имеющим линейную сложность, при условии, что хеш-значения распределены равномерно. Например, поиск в коллекции из 1000 элементов в среднем требует только 1 сравнение, а не 10, как в ассоциативном контейнере, и не 500, как в последовательном контейнере (см. раздел 2.2).

Подробное описание функций-членов, а также пример, демонстрирующий их применение, см. в разделе 7.7.2.

Таблица 7.51. Специальные операции поиска для неупорядоченных контейнеров

Операция	Действие
`c.count(`*val*`)`	Возвращает количество элементов, имеющих ключ/значение *val*
`c.find(`*val*`)`	Возвращает позицию первого элемента, имеющего ключ/значение *val* (или позицию `end()`, если таковой не найден)
`c.equal_range(`*val*`)`	Возвращает интервал со всеми элементами, ключи или значения которых равны *val* (т.е. первую и последнюю позиции, в которые можно вставить элемент со значением *val*)

Операции присваивания

Как показано в табл. 7.52, неупорядоченные контейнеры содержат лишь основные операции присваивания, характерные для всех контейнеров (см. раздел 7.1.2).

Таблица 7.52. Операции присваивания неупорядоченных контейнеров

Операция	Действие
`c = c2`	Присваивает все элементы контейнера `c2` контейнеру `c`
`c = rv`	Перемещает все элементы контейнера `rv` в массив `c` (по стандарту C++11)
`c = initlist`	Присваивает контейнеру `c` все элементы списка инициализации `initlist` (по стандарту C++11)
`c1.swap(c2)`	Обменивает элементы между контейнерами `c1` и `c2`
`swap(c1,c2)`	Обменивает элементы между контейнерами `c1` и `c2`

[16] Операторы == и != для неупорядоченных контейнеров в соответствии с документом TR1 не предусмотрены.

При выполнении этих операций оба контейнера должны иметь одинаковый тип. В частности, типы хеш-функций и критериев сравнения должны быть одинаковы, хотя сами хеш-функции могут быть разными. Если функции разные, их также можно присваивать и менять местами.

Функции для итераторов и доступа к элементам

Неупорядоченные контейнеры не предоставляют прямой доступ к элементам, поэтому необходимо использовать цикл `for` по интервалу (см. раздел 3.1.4) или итераторы. Поскольку гарантируется только однонаправленность итераторов (см. раздел 9.2.3), поддержка двунаправленных итераторов и итераторов произвольного доступа не обеспечивается (табл. 7.53). Таким образом, их нельзя использовать в алгоритмах, требующих только двунаправленных итераторов или итераторов произвольного доступа, например, в алгоритмах сортировки и случайного перемешивания.

В неупорядоченных (мульти)множествах все элементы рассматриваются как константные с точки зрения итератора. В неупорядоченных (мульти)отображениях ключ всех элементов считается константным. Это необходимо для того, чтобы гарантировать сохранность позиций элементов при изменении их значений. Несмотря на то что они не имеют конкретного порядка, значение определяет позицию сегмента в соответствии с текущей хеш-функцией. По этой причине к элементам нельзя применять никакие модифицирующие алгоритмы. Например, нельзя вызывать алгоритм `remove()`, потому что он "удаляет" элементы путем "замещения" их следующими за ними элементами (см. раздел 6.7.2). Для удаления элементов из неупорядоченных множеств и мультимножеств можно использовать только функции-члены, предусмотренные в этих контейнерах.

Таблица 7.53. Операции над итераторами в неупорядоченных контейнерах

Операция	Действие
`c.begin()`	Возвращает последовательный итератор, установленный на первый элемент
`c.end()`	Возвращает последовательный итератор, установленный на позицию, следующую за последним элементом
`c.cbegin()`	Возвращает константный последовательный итератор, установленный на первый элемент (по стандарту C++11)
`c.cend()`	Возвращает константный последовательный итератор, установленный на позицию, следующую за последним элементом (начиная со стандарта C++11)
`c.rbegin()`	Возвращает обратный итератор, установленный на первый элемент в обратном обходе
`c.rend()`	Возвращает обратный итератор, установленный на позицию, следующую за последним элементом в обратном обходе
`c.crbegin()`	Возвращает константный обратный итератор, установленный на первый элемент в обратном обходе (по стандарту C++11)
`c.crend()`	Возвращает константный обратный итератор, установленный на позицию, следующую за последним элементом в обратном обходе (по стандарту C++11)

В соответствии с определениями отображений и мультиотображений тип элементов неупорядоченных отображений и мультиотображений представляет собой класс `pair<const Key, T>`, т.е. для доступа к ключу и значению элемента необходимо использовать поля `first` и `second` (см. раздел 7.8.2).

```
std::unordered_map<std::string,float> coll;
...
for (auto& elem : coll) {
     std::cout << "key: " << elem.first << "\t"
               << "value: " << elem.second << std::endl;
}
...
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
     std::cout << "key: " << pos->first << "\t"
               << "value: " << pos->second << std::endl;
}
```

Попытка изменить значение ключа приведет к ошибке.

```
elem.first = "hello"; // ОШИБКА во время компиляции
pos->first = "hello"; // ОШИБКА во время компиляции
```

Однако изменение значения элемента не представляет проблемы, поскольку `elem` — это неконстантная ссылка, а тип значения также не является константным.

```
elem.second = 13.5; // OK
pos->second = 13.5; // OK
```

Если для манипуляций с элементами используются алгоритмы и лямбда-выражения, то необходимо явно объявить тип элемента.

```
std::unordered_map<std::string,int> coll;
...
// добавляем 10 к значению каждого элемента:
std::for_each (coll.begin(), coll.end(),
               [] (std::pair<const std::string,int>& elem) {
                   elem.second += 10;
               });
```

Для объявления типа элемента вместо типа

```
std::pair<const std::string,int>
```

можно использовать типы

```
std::unordered_map<std::string,int>::value_type
```

или

```
decltype(coll)::value_type
```

Законченный пример, демонстрирующий работу с отображениями, который также относится и к неупорядоченным отображениям, приведен в разделе 7.8.5.

Для изменения ключа элемента есть только одна возможность: заменить старый элемент новым элементом с тем же значением. Эта процедура описана для отображений в разделе 7.8.2.

Отметим, что неупорядоченные отображения можно использовать в качестве ассоциативных массивов, поэтому для доступа к их элементам можно применять операцию индексирования. Детали изложены в разделе 7.9.5.

Кроме того, существует дополнительный интерфейс итератора для обхода сегментов неупорядоченного контейнера. Детали см. в разделе 7.9.7.

Вставка и удаление элементов

В табл. 7.54 перечислены операции вставки и удаления элементов, предусмотренные для неупорядоченных контейнеров.

Как обычно при использовании библиотеки STL, программист должен гарантировать, что аргументы являются корректными. Итераторы должны ссылаться на корректные позиции, а начало интервала должно предшествовать его концу. В общем случае функции очистки не делают некорректными итераторы и ссылки на другие элементы. Однако функции-члены `insert()` и `emplace()` могут делать некорректными все итераторы при повторном хешировании, хотя ссылки на элементы всегда остаются правильными. Повторное хеширование выполняется тогда, когда после вставки очередного элемента общее количество элементов становится равным или превышает размер сегмента, умноженного на коэффициент заполнения (т.е. когда нарушаются гарантии, предоставленные коэффициентом максимального заполнения). Функции-члены `insert()` и `emplace()` не влияют на корректность ссылок, установленных на элементы контейнера.

Вставка и удаление выполняются быстрее, если при работе с несколькими элементами мы используем один вызов операции для всех элементов, а не несколько вызовов. Однако следует отметить, что гарантии исключений при работе с несколькими элементами оказываются ограничены (см. раздел 7.9.6).

Таблица 7.54. Операции вставки и удаления элементов неупорядоченных контейнеров

Операция	Действие
`c.insert(`*val*`)`	Вставляет копию значения *val* и возвращает позицию нового элемента и (для контейнеров `unordered_set` и `unordered_map`) — признак успешности операции
`c.insert(`*pos*`,` *val*`)`	Вставляет копию значения *val* и возвращает позицию нового элемента (параметр *pos* используется как подсказка, указывающая на позицию, с которой следует начинать поиск места для вставки)
`c.insert(`*beg*`,` *end*`)`	Вставляет копии всех элементов интервала [*beg*,*end*) (не возвращая ничего)
`c.insert(`*initlist*`)`	Вставляет копию всех элементов списка инициализации *initlist* (не возвращая ничего; по стандарту C++11)
`c.emplace(`*args*`...)`	Вставляет новый элемент, инициализированный списком аргументов args, и возвращает позицию нового элемента и (для множеств) — признак успешности операции (по стандарту C++11)

Окончание табл. 7.54

Операция	Действие
`c.emplace_hint(pos,args...)`	Вставляет новый элемент, инициализированный списком аргументов args, и возвращает позицию нового элемента (параметр *pos* используется как подсказка, указывающая на позицию, с которой следует начинать поиск места для вставки)
`c.erase(val)`	Удаляет все элементы, равные *val*, и возвращает количество удаленных элементов
`c.erase(pos)`	Удаляет элемент, занимающий позицию итератора *pos*, и возвращает следующую позицию (до появления стандарта C++11 не возвращал ничего)
`c.erase(beg,end)`	Удаляет все элементы интервала [*beg*,*end*) и возвращает следующую позицию (до появления стандарта C++11 не возвращал ничего)
`c.clear()`	Удаляет все элементы (опустошает контейнер)

Рассмотрим различия между типами значений, возвращаемых функциями вставки `insert()` и `emplace()`.

- **Неупорядоченные множества** имеют следующий интерфейс:

```
pair<iterator,bool> insert(const value_type& val);
iterator            insert(iterator posHint,
                           const value_type& val);
template <typename... Args>
pair<iterator,bool> emplace(Args&&... args);
template <typename... Args>
iterator            emplace_hint(const_iterator posHint,
                                 Args&&... args);
```

- **Неупорядоченные мультимножества**

```
iterator            insert(const value_type& val);
iterator            insert(iterator posHint,
                           const value_type& val);
template <typename... Args>
iterator            emplace(Args&&... args);
template <typename... Args>
iterator            emplace_hint(const_iterator posHint,
                                 Args&&... args);
```

Разница между типами возвращаемых значений объясняется тем, что мультимножества допускают дубликаты, а множества — нет. Таким образом, вставка элемента в множество может завершиться сбоем, если оно уже содержит элемент с таким же значением. По этой причине тип возвращаемого значения для множества представляет собой структуру `pair` (структура `pair` обсуждается в разделе 5.1.1).

1. Член `second` структуры `pair` указывает, успешно ли прошла операция вставки.
2. Член `first` структуры `pair` содержит позицию вновь вставленного элемента или позицию ранее существовавшего элемента.

Во всех остальных случаях функции возвращают позицию нового или существующего элемента, если множество уже содержит элемент с таким же значением. Примеры таких интерфейсом см. в разделе 7.7.2.

При вставке пары "ключ–значение" в неупорядоченные отображения и мультиотображения следует помнить, что ключ считается константой. Таким образом, следует предоставить корректный тип, или неявные, или явные преобразования типов.

По стандарту C++11 наиболее удобным способом вставки элементов является их передача в виде списка инициализации, в котором первая запись является ключом, а вторая — значением.

```
std::unordered_map<std::string,float> coll;
...
coll.insert({"otto",22.3});
```

Существуют три альтернативных способа передачи значения в неупорядоченное отображение или мультиотображение, которые уже были описаны для обычных отображений и мультиотображений (см. раздел 7.8.2).

1. **Использование типа `value_type`.**

```
std::unordered_map<std::string,float> coll;
...
coll.insert(std::unordered_map<std::string,float>::value_type
            ("otto",22.3));
coll.insert(decltype(coll)::value_type("otto",22.3));
```

2. **Использование типа `pair<>`.**

```
std::unordered_map<std::string,float> coll;
...
coll.insert(std::pair<std::string,float>("otto",22.3));
coll.insert(std::pair<const std::string,float>("otto",22.3));
```

3. **Использование функции `make_pair()`.**

```
std::unordered_map<std::string,float> coll;
...
coll.insert(std::make_pair("otto",22.3));
```

Рассмотрим простой пример вставки элемента в неупорядоченное отображение, в котором производится проверка его успешности:

```
std::unordered_map<std::string,float> coll;
...
if (coll.insert(std::make_pair("otto",22.3)).second) {
    std::cout << "OK, could insert otto/22.3" << std::endl;
}
else {
    std::cout << "Oops, could not insert otto/22.3 "
              << "(key otto already exists)" << std::endl;
}
```

Снова отметим, что при использовании функции `emplace()` для вставки нового элемента путем передачи значений для его создания необходимо передавать два списка аргументов: один для ключей, а другой — для значений. Удобнее всего это сделать следующим образом:

```
std::unordered_map<std::string,std::complex<float>> m;
m.emplace(std::piecewise_construct,  // передаем в качестве
                                     // аргументов кортежи
          std::make_tuple("hello"),  // элементы для ключей
          std::make_tuple(3.4,7.8)); // элементы для значения
```

Подробности создания пар по частям см. в разделе 5.1.1.

Неупорядоченные отображения предоставляют удобный способ для вставки и задания элементов с помощью операции индексации (см. раздел 7.9.5).

Для того чтобы удалить элемент с заданным значением, можно просто вызвать функцию-член `erase()`.

```
std::unordered_set<Elem> coll;
...
// удаляем все элементы с заданным значением
coll.erase(value);
```

Имя этой функции-члена отличается от имени функции-члена `remove()`, предусмотренной для списка (обсуждение функции `remove()` см. в разделе 7.5.2). Функция `erase()` работает иначе, потому что возвращает количество удаленных элементов. При вызове для неупорядоченных отображений она возвращает только `0` или `1`.

Если неупорядоченные мультимножества или мультиотображения содержат дубликаты и вы хотите удалить только первый элемент из этих дубликатов, функцию-член `erase()` вызывать нельзя. Вместо нее следует выполнить следующий код:

```
std::unordered_multimap<Key,T> coll;
...
// удаляем первый элемент с заданным значением
auto pos = coll.find(value);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

Здесь следует использовать функцию-член `find()`, потому что она работает быстрее, чем алгоритм `find()` (см. раздел 7.3.2).

При удалении элементов необходимо быть осторожным и не спилить сук, на котором сидите (подробное описание этой проблемы см. в разделе 7.8.2).

7.9.4. Интерфейс сегментов

Существует возможность доступа к отдельным сегментам с помощью специального интерфейса, для того чтобы получить информацию о внутреннем состоянии всех хеш-таблиц. В табл. 7.55 перечислены операции для прямого доступа к сегментам.

Таблица 7.55. Интерфейс операций над сегментами неупорядоченных множеств и мультимножеств

Операция	Действие
`c.bucket_count()`	Возвращает текущее количество сегментов
`c.bucket(`*val*`)`	Возвращает индекс сегмента, в котором может (или мог бы) находиться элемент со значением *val*
`c.bucket_size(`*buckidx*`)`	Возвращает количество элементов в сегменте с индексом *buckidx*
`c.begin(`*buckidx*`)`	Возвращает однонаправленный итератор, установленный на первый элемент сегмента с индексом *buckidx*
`c.end(`*buckidx*`)`	Возвращает однонаправленный итератор, установленный на позицию, следующую за последним элементом сегмента с индексом *buckidx*
`c.cbegin(`*buckidx*`)`	Возвращает константный однонаправленный итератор, установленный на первый элемент сегмента с индексом *buckidx*
`c.cend(`*buckidx*`)`	Возвращает константный однонаправленный итератор, установленный на позицию, следующую за последним элементом сегмента с индексом *buckidx*

Пример использования интерфейса сегмента для проверки внутренней структуры неупорядоченного контейнера см. в разделе 7.9.7.

7.9.5. Использование неупорядоченных отображений в качестве ассоциативных массивов

Как и отображения, неупорядоченные отображения предоставляют оператор индексирования и соответствующую функцию-член `at()` (см. табл. 7.56) для непосредственного доступа к элементу.

Таблица 7.56. Прямой доступ к элементам неупорядоченных отображений

Операция	Действие
`c[`*key*`]`	Вставляет элемент с ключом *key* (если такового еще не было в контейнере) и возвращает ссылку на значение элемента с ключом *key* (только для неконстантных неупорядоченных отображений)
`c.at(`*key*`)`	Возвращает ссылку на значение элемента с ключом *key* (по стандарту C++11)

Функция-член `at()` возвращает значение элемента с заданным ключом и генерирует исключение типа `out_of_range`, если такого элемента нет.

В операторе `[ ]` индекс также является ключом, используемым для идентификации элемента. Это значит, что в операторе `[ ]` индекс может иметь любой тип, а не только целочисленный. Такой интерфейс является интерфейсом так называемого *ассоциативного массива*.

Тип индекса в операторе `[ ]` для ассоциативных массивов — не единственное отличие от обычных массивов. Кроме того, индекс не бывает неправильным. Если ключом является индекс, для которого не существует ни одного элемента, в отображение автоматически

вставляется новый элемент. Значение нового элемента инициализируется конструктором его типа, заданным по умолчанию. Таким образом, для использования этой возможности нельзя использовать тип значения, не имеющего конструктора, заданного по умолчанию. Отметим, что все элементарные типы имеют конструктор по умолчанию, инициализирующие их значения нулями (см. раздел 3.2.1).

В разделе 7.8.3 подробно описаны преимущества и недостатки такого контейнерного интерфейса. Примеры программы (в частности, использующие отображения с такими же интерфейсами) см. в разделах 6.2.4 и 7.8.5.

7.9.6. Обработка исключений

Неупорядоченные контейнеры основаны на узлах, поэтому любой сбой при создании узла просто оставляет контейнер в неизменном состоянии. Однако при этом следует учитывать возможность повторного хеширования. По этой причине на все неупорядоченные контейнеры распространяются следующие гарантии.

- Вставка одного элемента выполняется по принципу "все или ничего", при условии, что хеш-функция и критерий эквивалентности не генерируют исключений. Такие образом, если они не генерируют исключений, операции либо выполняются успешно, либо не меняют состояние контейнера.
- Функция-член `erase()` не генерирует исключений, при условии, что операции копирования/удаления (конструкторы и операции присваивания) элементов не генерируют исключений.
- Функция-член `clear()` не генерирует исключений.
- Функция-член `swap()` не генерирует исключений, при условии, что копирующий конструктор или копирующая операция присваивания хеш-функции или критерий эквивалентности не генерируют исключений.
- Функция-член `rehash()` выполняется по принципу "все или ничего", при условии, что хеш-функция и критерий эквивалентности не генерируют исключений. Такие образом, если они не генерируют исключений, операции либо выполняются успешно, либо не меняют состояние контейнера.

Общее обсуждение обработки исключений в библиотеке STL изложено в разделе 6.12.2.

7.9.7. Примеры использования неупорядоченных контейнеров

Следующая программа демонстрирует возможности неупорядоченных контейнеров на примере неупорядоченных множеств:

```
// cont/unordset1.cpp

#include <unordered_set>
#include <numeric>
#include "print.hpp"
```

```
using namespace std;

int main()
{
   // создаем и инициализируем неупорядоченное множество
   unordered_set<int> coll = { 1,2,3,5,7,11,13,17,19,77 };

   // выводим элементы на экран
   // - элементы следуют в произвольном порядке
   PRINT_ELEMENTS(coll);

   // вставляем дополнительные элементы
   // - эта операция может вызвать повторное хеширование
   // и создать другой порядок
   coll.insert({-7,17,33,-11,17,19,1,13});
   PRINT_ELEMENTS(coll);

   // удаляем элемент с заданным значением
   coll.erase(33);

   // вставляем сумму всех существующих значений
   coll.insert(accumulate(coll.begin(),coll.end(),0));
   PRINT_ELEMENTS(coll);

   // проверяем, если ли в множестве значение 19
   if (coll.find(19) != coll.end()) {
       cout << "19 is available" << endl;
   }

   // удаляем все отрицательные значения
   unordered_set<int>::iterator pos;
   for (pos=coll.begin(); pos!= coll.end(); ) {
        if (*pos < 0) {
            pos = coll.erase(pos);
        }
      else {
            ++pos;
      }
   }
   PRINT_ELEMENTS(coll);
}
```

При выполнении вставок, удаления и поиска элемента с заданным значением неупорядоченные контейнеры обеспечивают максимальную производительность программ, потому что все эти операции имеют амортизированную константную сложность.

Однако мы не можем делать никаких предположений о порядке элементов. Например, программа может вывести на экран следующие строки:

```
77 11 1 13 2 3 5 17 7 19
-11 1 2 3 -7 5 7 77 33 11 13 17 19
-11 1 2 3 -7 5 7 77 11 13 17 19 137
19 is available
1 2 3 5 7 77 11 13 17 19 137
```

Для того чтобы сделать что-то еще — например, сложить значения элементов в контейнере иди найти и удалить все отрицательные значения, — необходимо выполнить обход всех элементов (либо непосредственно с помощью итераторов, либо косвенно с помощью цикла `for` по интервалу).

При использовании неупорядоченного мультимножества, а не множества разрешаются дубликаты. Рассмотрим следующую программу:

```
// cont/unordmultiset1.cpp

#include <unordered_set>
#include "print.hpp"
using namespace std;

int main()
{
   // создаем и инициализируем, расширяем и выводим на экран
   // неупорядоченные мультимножества
   unordered_multiset<int> coll = { 1,2,3,5,7,11,13,17,19,77 };
   coll.insert({-7,17,33,-11,17,19,1,13});
   PRINT_ELEMENTS(coll);

   // удаляем все элементы с заданным значением
   coll.erase(17);

   // удаляем один элемент с заданным значением
   auto pos = coll.find(13);
   if (pos != coll.end()) {
      coll.erase(pos);
   }
   PRINT_ELEMENTS(coll);
}
```

Она может вывести на экран следующие строки:

```
33 19 19 17 17 17 77 11 7 -7 5 3 13 13 2 -11 1 1
33 19 19 77 11 7 -7 5 3 13 2 -11 1 1
```

Пример создания собственных хеш-функций и критерия эквивалентности

Следующий пример демонстрирует способ определения хеш-функции и критерия эквивалентности для типа `Customer`, используемого как тип элемента неупорядоченного массива:

```
// cont/hashfunc1.cpp

#include <unordered_set>
#include <string>
#include <iostream>
#include "hashval.hpp"
#include "print.hpp"
```

```
using namespace std;

class Customer {
   private:
      string fname;
      string lname;
      long no;
   public:
      Customer (const string& fn, const string& ln, long n)
         : fname(fn), lname(ln), no(n) {}
      friend ostream& operator << (ostream& strm, const Customer& c) {
         return strm << "[" << c.fname << "," << c.lname << ","
                     << c.no << "]";
      }
      friend class CustomerHash;
      friend class CustomerEqual;
   };

   class CustomerHash
   {
      public:
         std::size_t operator() (const Customer& c) const {
              return hash_val(c.fname,c.lname,c.no);
   }
};

class CustomerEqual
{
   public:
      bool operator() (const Customer& c1, const Customer& c2) const {
         return c1.no == c2.no;
   }
};

int main()
{
   // неупорядоченное множество с пользовательской хеш-функцией
   // и критерием эквивалентности
   unordered_set<Customer,CustomerHash,CustomerEqual> custset;

   custset.insert(Customer("nico","josuttis",42));
   PRINT_ELEMENTS(custset);
}
```

Программа выводит на экран такие строки:

```
[nico,josuttis,42]
```

Здесь используется вспомогательная функция-член `hash_val()`, позволяющая работать с произвольным количеством элементов разного типа и описанная в разделе 7.9.2.

Легко видеть, что функция эквивалентности не обязательно вычисляет те же значения, что и хеш-функция. Однако она должна гарантировать, что одинаковые значения, соответствующие критерию эквивалентности, порождают одинаковые хеш-значения (в нашем примере это косвенно основано на предположении об уникальности номеров потребителей).

Если функция эквивалентности не указывается, то объявление объекта `custset` может иметь следующий вид:

```
std::unordered_set<Customer,CustomerHash> custset;
```

а в качестве критерия эквивалентности используется оператор ==, который следует определить в классе `Customer`.

В качестве хеш-функции можно использовать обычную функцию. Однако, если функция передается как аргумент конструктора, необходимо передать начальный счетчик сегмента и задать соответствующий указатель на функцию в качестве второго шаблонного параметра (детали см. в разделе 7.9.2).

Использование лямбда-функций в качестве хеш-функции и критерия эквивалентности

Хеш-функцию и критерий эквивалентности можно задать даже с помощью лямбда-функций. Рассмотрим пример:

```
// cont/hashfunc2.cpp

#include <string>
#include <iostream>
#include <unordered_set>
#include "hashval.hpp"
#include "print.hpp"
using namespace std;

class Customer {
   private:
      string fname;
      string lname;
      long no;
   public:
      Customer (const string& fn, const string& ln, long n)
         : fname(fn), lname(ln), no(n) {
      }

      string firstname() const {
         return fname;
      };

      string lastname() const {
         return lname;
      };

      long number() const {
         return no;
      };

      friend ostream& operator << (ostream& strm, const Customer& c) {
         return strm << "[" << c.fname << "," << c.lname << ","
                     << c.no << "]";
      }
```

```
};

int main()
{
   // лямбда-функция для пользовательской хеш-функции
   auto hash = [] (const Customer& c) {
      return hash_val(c.firstname(),c.lastname(),c.number());
   };

   // лямбда-функция для пользовательского критерия эквивалентности
   auto eq = [] (const Customer& c1, const Customer& c2) {
      return c1.number() == c2.number();
   };

   // создаем неупорядоченное множество с пользовательскими
   // хеш-функцией и критерием эквивалентности
   unordered_set<Customer,
                 decltype(hash),decltype(eq)> custset(10,hash,eq);
   custset.insert(Customer("nico","josuttis",42));
   PRINT_ELEMENTS(custset);
}
```

Для получения типа лямбда-функции необходимо использовать ключевое слово decltype, чтобы иметь возможность передать шаблонный аргумент в объявление неупорядоченного контейнера. Причина заключается в том, что для лямбда-функций не определены конструкторы по умолчанию и операция присваивания. Следовательно, лямбда-функции необходимо передавать в конструктор. Это возможно только с помощью второго и третьего аргументов. Таким образом, мы должны задать начальный размер сегмента (в данном случае равным 10).

Пример использования интерфейса сегмента

Следующий пример демонстрирует использование интерфейса сегмента для проверки внутреннего состояния неупорядоченного контейнера (см. раздел 7.9.4). Функция printHashTableState() выводит на экран все состояние, включая подробную структуру неупорядоченного контейнера.

```
// cont/buckets.hpp

#include <iostream>
#include <iomanip>
#include <utility>
#include <iterator>
#include <typeinfo>

// обобщенная функция для вывода пар (элементом отображения)
template <typename T1, typename T2>
std::ostream& operator << (std::ostream& strm, const std::pair<T1,T2>& p)
{
   return strm << "[" << p.first << "," << p.second << "]";
}

template <typename T>
```

```
void printHashTableState (const T& cont)
{
   // основные данные о структуре
   std::cout << "size:            " << cont.size() << "\n";
   std::cout << "buckets:         " << cont.bucket_count() << "\n";
   std::cout << "load factor:     " << cont.load_factor() << "\n";
   std::cout << "max load factor: " << cont.max_load_factor() << "\n";

   // категория итератора
   if (typeid(typename std::iterator_traits
                <typename T::iterator>::iterator_category)
       == typeid(std::bidirectional_iterator_tag)) {
       std::cout << "chaining style: doubly-linked" << "\n";
   }
   else {
       std::cout << "chaining style: singly-linked" << "\n";
   }

   // количество элементов в сегменте:
   std::cout << "data: " << "\n";
   for (auto idx=0; idx != cont.bucket_count(); ++idx) {
         std::cout << " b[" << std::setw(2) << idx << "]: ";
         for (auto pos=cont.begin(idx); pos != cont.end(idx); ++pos) {
              std::cout << *pos << " ";
         }
         std::cout << "\n";
   }
   std::cout << std::endl;
}
```

Этот заголовочный файл можно использовать, например, для вывода на экран внутренней структуры неупорядоченного множества:

```
// cont/unordinspect1.cpp

#include <unordered_set>
#include <iostream>
#include "buckets.hpp"

int main()
{
   // создаем и инициализируем неупорядоченное множество
   std::unordered_set<int> intset = { 1,2,3,5,7,11,13,17,19 };
   printHashTableState(intset);

   // вставляем дополнительные значения
   // (может вызвать повторное хеширование)
   intset.insert({-7,17,33,4});
   printHashTableState(intset);
}
```

Сравнение первого и второго вызовов функции `printHashTableState()` показывает, что программа может дать следующий вывод (детали зависят от конкретной структуры и стратегии повторного хеширования, принятой в стандартной библиотеке).

```
Size:            9
buckets:         11
load factor:     0.818182
max load factor: 1
chaining style: singly-linked
data:
b[ 0]: 11
b[ 1]: 1
b[ 2]: 13 2
b[ 3]: 3
b[ 4]:
b[ 5]: 5
b[ 6]: 17
b[ 7]: 7
b[ 8]: 19
b[ 9]:
b[10]:
```

```
size:            9
buckets:         11
load factor:     0.818182
max load factor: 1
chaining style: singly-linked
data:
b[ 0]: 11
b[ 1]: 1
b[ 2]: 13 2
b[ 3]: 3
b[ 4]:
b[ 5]: 5
b[ 6]: 17
b[ 7]: 7
b[ 8]: 19
b[ 9]:
b[10]:

size:            12
buckets:         23
load factor:     0.521739
max load factor: 1
chaining style: singly-linked
data:
b[ 0]:
b[ 1]: 1
b[ 2]: 2
b[ 3]: 3
b[ 4]: 4
b[ 5]: 5 -7
b[ 6]:
b[ 7]: 7
b[ 8]:
b[ 9]:
b[10]: 33
b[11]: 11
b[12]:
b[13]: 13
b[14]:
b[15]:
b[16]:
b[17]: 17
b[18]:
b[19]: 19
b[20]:
b[21]:
b[22]:
```

Следующая программа, представляющая собой другой пример применения интерфейса сегментов, создает словарь строк, отображаемых в другие строки (сравните этот пример с соответствующей версией отображений, описанных в разделе 7.8.5):

```
// cont/unordmultimap1.cpp

#include <unordered_map>
#include <string>
#include <iostream>
#include <utility>
```

```
#include "buckets.hpp"
using namespace std;

int main()
{
   // создаем и инициализируем неупорядоченное мультиотображение как словарь
   std::unordered_multimap<string,string> dict = {
                {"day","Tag"},
                {"strange","fremd"},
                {"car","Auto"},
                {"smart","elegant"},
                {"trait","Merkmal"},
                {"strange","seltsam"}
   };
   printHashTableState(dict);

   // вставляем дополнительные значения
   // (может вызвать повторное хеширование)
   dict.insert({{"smart","raffiniert"},
               {"smart","klug"},
               {"clever","raffiniert"}
   });
   printHashTableState(dict);
   // изменяем коэффициент максимальной нагрузки
   // (может вызывать повторное хеширование)
   dict.max_load_factor(0.7);
   printHashTableState(dict);
}
```

Результат работы программы зависит от реализации. Например, он может быть таким, как показано ниже (масштаб выбран с учетом ширины страницы).

```
size:                 6
buckets:              7
current load factor:
0.857143
max load factor:      1
chaining style: singly
data:
b[ 0]: [day,Tag]
b[ 1]: [car,Auto]
b[ 2]:
b[ 3]: [smart,elegant]
b[ 4]:
b[ 5]: [trait,Merkmal]
b[ 6]: [strange,fremd]
        [strange,seltsam]
```

```
size:                 9
buckets:              11
current load factor:
0.818182
max load factor:      1
chaining style: singly
data:
b[ 0]: [smart,elegant]
        [smart,raffiniert]
         [smart,klug]
b[ 1]:
b[ 2]:
b[ 3]:
b[ 4]:
b[ 5]: [clever,raffiniert]
b[ 6]: [strange,fremd]
        [strange,seltsam]
b[ 7]:
b[ 8]:
b[ 9]: [trait,Merkmal]
[car,Auto]
b[10]: [day,Tag]
```

```
size:                 9
buckets:              17
current load factor:
0.529412
max load factor:      0.7
chaining style: singly
data:
b[ 0]:
b[ 1]:
b[ 2]:
b[ 3]:
b[ 4]: [car,Auto]
b[ 5]:
b[ 6]: [smart,elegant]
        [smart,raffiniert]
         [smart,klug]
b[ 7]:
b[ 8]: [day,Tag]
b[ 9]: [clever,raffiniert]
b[10]:
b[11]: [trait,Merkmal]
b[12]:
b[13]:
b[14]:
b[15]: [strange,fremd]
        [strange,seltsam]
b[16]:
```

На другой платформе результат может оказаться иным (масштаб выбран с учетом ширины страницы).

```
size:                6
buckets:             11
current load factor:
0.545455
max load factor:     1
chaining style: singly
data:
b[ 0]:
b[ 1]:
b[ 2]: [trait,Merkmal]
       [car,Auto]
b[ 3]: [day,Tag]
b[ 4]:
b[ 5]:
b[ 6]:
b[ 7]:
b[ 8]: [smart,elegant]
b[ 9]: [strange,seltsam]
       [strange,fremd]
b[10]:
```

```
size:                9
buckets:             11
current load factor:
0.818182
max load factor:     1
chaining style: singly
data:
b[ 0]:
b[ 1]:
b[ 2]: [clever,raffiniert]
       [trait,Merkmal]
       [car,Auto]
b[ 3]: [day,Tag]
b[ 4]:
b[ 5]:
b[ 6]:
b[ 7]:
b[ 8]: [smart,elegant]
       [smart,raffiniert]
       [smart,klug]
b[ 9]: [strange,seltsam]
       [strange,fremd]
b[10]:
```

```
size:                9
buckets:             13
current load factor:
0.692308
max load factor:     0.7
chaining style: singly
data:
b[ 0]:
b[ 1]: [day,Tag]
b[ 2]:
b[ 3]:
b[ 4]: [smart,elegant]
       [smart,raffiniert]
       [smart,klug]
       [car,Auto]
b[ 5]:
b[ 6]:
b[ 7]:
b[ 8]:
b[ 9]: [clever,raffiniert]
b[10]: [strange,seltsam]
       [strange,fremd]
b[11]:
b[12]: [trait,Merkmal]
       [clever,raffiniert]
```

Отметим, что в любом случае повторное хеширование сохраняет относительный порядок следования эквивалентных элементов. Однако порядок следования эквивалентных элементов не соответствует порядку их вставки.

7.10. Другие контейнеры STL

Библиотека STL — это каркас. Помимо стандартных контейнерных классов, она позволяет использовать другие структуры данных в качестве контейнеров. В роли контейнера STL можно использовать строки или обычные массивы, а можно написать специальные контейнерные классы, удовлетворяющие особые потребности. Это позволяет использовать преимущество алгоритмов, таких как сортировка или слияние, настроенных на пользовательский тип. Такой каркас является хорошим примером *принципа открытости/закрытости*: *открытость* для расширения, *закрытость* для модификаций[17].

Существуют три способа создать контейнеры в стиле библиотеки STL.

1. **Инвазивный подход**[18]. Программист просто предоставляет интерфейс, подразумеваемый контейнером из библиотеки STL. В частности, необходимы обычные функции-члены контейнеров, такие как `begin()` и `end()`. Этот подход является инвазивным, потому что требует, чтобы контейнер был написан определенным образом.
2. **Неинвазивный подход**[18]. Программист пишет или предоставляет специальный итератор, который используется как интерфейс между алгоритмом и специальными

[17] Впервые я услышал о *принципе открытости/закрытости* от Роберта Мартина (Robert C. Martin), который сам узнал о нем от Бертрана Мейера (Bertrand Meyer).

[18] Вместо терминов *инвазивный* и *неинвазивный* иногда используются термины *интрузивный* и *неинтрузивный*.

контейнерами. Этот подход является неинвазивным. Для него требуется лишь возможность перебора элементов контейнера, которую так или иначе предоставляет любой контейнер.

3. **Подход, основанный на использовании оболочки.** Комбинируя два описанных выше подхода, программист пишет класс-оболочку, инкапсулирующий структуру данных и STL-подобный интерфейс.

В этом подразделе сначала обсуждаются строки как стандартный контейнер, являющийся примером инвазивного подхода. Затем обсуждается важный стандартный контейнер, использующий неинвазивный подход: обычный массив в стиле языка C. Однако для доступа к элементам обычного контейнера можно использовать и оболочку.

Создавая контейнер STL, можно также параметризовать разные распределители памяти. Стандартная библиотека C++ содержит несколько специальных функций и классов для программирования распределителей памяти и ее освобождения. Детали изложены в разделе 19.3.

7.10.1. Строки как контейнеры STL

Классы строк в стандартной библиотеке C++ (рассматриваемые в главе 13) являются примером инвазивного подхода при написании контейнеров STL. Строки можно рассматривать как контейнеры символов. Символы в строке образуют последовательность, которую необходимо пройти, чтобы обработать отдельные символы. Таким образом, стандартные классы строк обеспечивают интерфейс контейнера STL. Они содержат функции-члены `begin()` и `end()`, возвращающие итераторы прямого доступа для обхода строки. Классы строк также содержат операции над итераторами и адаптеры итераторов. Например, функция-член `push_back()` позволяет использовать механизмы вставки в конец строки.

Обработка строки с точки зрения библиотеки STL выглядит немного необычно. Как правило, строка обрабатывается как целостный объект (т.е. строки передаются, копируются и присваиваются целиком). Однако, если необходимо обработать отдельный символ, целесообразно использовать алгоритмы STL. Например, с помощью потоковых итераторов ввода можно считывать символы или переводить их из одного регистра клавиатуры в другой. Кроме того, с помощью алгоритмов STL можно использовать специальный критерий сравнения строк. Стандартный интерфейс строки не предусматривает такой возможности.

Более подробно строки и их возможности рассматриваются в разделе 13.2.14.

7.10.2. Обычные массивы в стиле языка C как контейнеры STL

В качестве контейнеров STL можно использовать обычные массивы в стиле языка C. Правда, эти массивы не являются классами, поэтому у них нет функций-членов, таких как `begin()` и `end()`, и программист не может определить их самостоятельно. В такой ситуации нужен неинвазивный подход или класс-оболочка.

Использование неинвазивного подхода не составляет труда. Нужен лишь объект для обхода элементов массива с помощью интерфейса итератора STL. Такие итераторы уже существуют — это обычные указатели. Проектировщики библиотеки STL решили использовать

интерфейс указателя для итераторов, чтобы указатели могли заменять итераторы. Это вновь демонстрирует нам универсальную концепцию чистой абстракции: если нечто *ведет себя* как итератор, то оно *является* итератором. Фактически указатели — это итераторы произвольного доступа (см. раздел 9.2.5). Следующий пример демонстрирует использование массивов в стиле языка C в качестве контейнеров STL в соответствии со стандартом C++11:

```
// cont/cstylearray1.cpp

#include <iterator>
#include <vector>
#include <iostream>

int main()
{
   int vals[] = { 33, 67, -4, 13, 5, 2 };

   // применяем функции begin() и end() к обычным массивам в стиле языка C
   std::vector<int> v(std::begin(vals), std::end(vals));

   // используем глобальные функции begin() и end() для контейнеров
   std::copy (std::begin(v), std::end(v),
              std::ostream_iterator<int>(std::cout," "));
   std::cout << std::endl;
}
```

Здесь используется вспомогательная функция, определенная в заголовочном файле `<iterator>` и в каждом контейнерном заголовке, что позволяет применять глобальные функции `begin()` и `end()` к обычным массивам в стиле языка C. Легко видеть, что для любого обычного массива `vals` в стиле языка C функции `std::begin()` и `std::end()` возвращают его начало и конец, которые можно использовать в рамках библиотеки STL.

```
int vals[] = { 33, 67, -4, 13, 5, 2 };
std::begin(vals) // возвращает указатель vals
std::end(vals)   // возвращает указатель vals+NumOfElementsIn(vals)
```

Эти функции также перегружены, так что можно использовать контейнеры STL или все классы, имеющие функции-члены `begin()` и `end()`.

```
std::vector<int> v;
std::begin(v)    // возвращает v.begin()
std::end(v)      // возвращает v.end()
```

Результат работы этой программы выглядит следующим образом:

```
33 67 -4 13 5 2
```

До появления стандарта C++11 алгоритмам необходимо было передавать обычные указатели, потому что глобальных функций `begin()` и `end()` не было. Рассмотрим пример:

```
// cont/cstylearray1old.cpp

#include <iostream>
#include <algorithm>
#include <functional>
#include <iterator>
using namespace std;
```

```
int main()
{
   int coll[] = { 5, 6, 2, 4, 1, 3 };

   // возводим в квадрат все элементы
   transform (coll, coll+6,                  // первый источник
              coll,                          // второй источник
              coll,                          // приемник
              multiplies<int>());            // операция

   // применяем алгоритм sort, начиная со второго элемента
   sort (coll+1, coll+6);

   // выводим на экран все элементы
   copy (coll, coll+6,
         ostream_iterator<int>(cout," "));
   cout << endl;
}
```

При передаче конца массива следует быть внимательным. В данном случае конец массива соответствует адресу `coll+6`. И, как обычно, следует гарантировать, что конец интервала — это позиция, следующая за последним элементом.

Результат работы этой программы выглядит следующим образом:

```
25 1 4 9 16 36
```

Дополнительные примеры использования обычных массивов в стиле языка C приведены в разделах 11.7.2 и 11.10.2.

7.11. Реализация семантики ссылок

В принципе контейнерные классы STL реализуют семантику значений, а не ссылок. Контейнеры создают внутренние копии элементов, которые они содержат, и возвращают копии этих элементов. Аргументы за и против этого подхода и его последствия обсуждаются в разделе 6.11.2. Подводя итоги, заметим, что если программист желает реализовать семантику ссылок в контейнерах STL — либо потому, что копирование элементов требует избыточных затрат ресурсов, либо потому, что идентичные элементы используются разными коллекциями, — необходимо использовать класс интеллектуальных указателей, позволяющий избежать возможных ошибок. Кроме того, возможно также использование оболочки ссылок.

Использование разделяемых указателей

Как указано в разделе 5.2, стандартная библиотека C++ содержит разные классы интеллектуальных указателей. Для объектов, разделяемых несколькими контейнерами, подходит класс `shared_ptr<>`. Его использование можно проиллюстрировать следующим примером:

```
// cont/refsem1.cpp

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

```
#include <set>
#include <deque>
#include <algorithm>
#include <memory>

class Item {
   private:
      std::string name;
      float price;
   public:
      Item (const std::string& n, float p = 0) : name(n), price(p) {
      }
      std::string getName () const {
         return name;
      }

      void setName (const std::string& n) {
         name = n;
      }

      float getPrice () const {
         return price;
      }

      float setPrice (float p) {
         price = p;
      }
};

template <typename Coll>
void printItems (const std::string& msg, const Coll& coll)
{
   std::cout << msg << std::endl;
   for (const auto& elem : coll) {
        std::cout << ' ' << elem->getName() << ": "
                  << elem->getPrice() << std::endl;
   }
}

int main()
{
   using namespace std;

   // две разные коллекции совместно владеют объектами класса Items
   typedef shared_ptr<Item> ItemPtr;
   set<ItemPtr> allItems;
   deque<ItemPtr> bestsellers;

   // вставляем объекты в коллекции
   // - бестселлеры принадлежат обеим коллекциям
   bestsellers = { ItemPtr(new Item("Kong Yize",20.10)),
                   ItemPtr(new Item("A Midsummer Night's Dream",14.99)),
                   ItemPtr(new Item("The Maltese Falcon",9.88)) };
   allItems = { ItemPtr(new Item("Water",0.44)),
                ItemPtr(new Item("Pizza",2.22)) };
```

```
    allItems.insert(bestsellers.begin(),bestsellers.end());

    // выводим на экран содержимое обеих коллекций
    printItems ("bestsellers:", bestsellers);
    printItems ("all:", allItems);
    cout << endl;

    // удваиваем цену каждого бестселлера
    for_each (bestsellers.begin(), bestsellers.end(),
              [] (shared_ptr<Item>& elem) {
                  elem->setPrice(elem->getPrice() * 2);
              });

    // заменяем второй бестселлер первым элементом с названием "Pizza"
    bestsellers[1] = *(find_if(allItems.begin(),allItems.end(),
                       [] (shared_ptr<Item> elem) {
                           return elem->getName() == "Pizza";
                       }));

    // устанавливаем цену первого бестселлера
    bestsellers[0]->setPrice(44.77);

    // выводим на экран содержимое обеих коллекций
    printItems ("bestsellers:", bestsellers);
    printItems ("all:", allItems);
}
```

Эта программа выводит на экран следующий результат:

```
bestsellers:
 Kong Yize: 20.1
 A Midsummer Night's Dream: 14.99
 The Maltese Falcon: 9.88
all:
 Kong Yize: 20.1
 A Midsummer Night's Dream: 14.99
 The Maltese Falcon: 9.88
 Water: 0.44
 Pizza: 2.22

bestsellers:
 Kong Yize: 44.77
 Pizza: 2.22
 The Maltese Falcon: 19.76
all:
 Kong Yize: 44.77
 A Midsummer Night's Dream: 29.98
 The Maltese Falcon: 19.76
 Water: 0.44
 Pizza: 2.22
```

Отметим, что использование класса `shared_ptr<>` значительно усложняет работу. Например, функция `find()` для множеств, которая ищет элементы с одинаковыми значениями, теперь будет сравнивать внутренние указатели, возвращаемые операцией `new`.

```
allItems.find(ItemPtr(new Item("Pizza",2.22))) // не может быть успешным
```

Следовательно, здесь необходимо использовать алгоритм `find_if()`.

Если вызвать вспомогательную функцию, сохраняющую где-то первый элемент коллекции (объект класса `ItemPtr`), то значение, на которое он ссылается, останется корректным, даже если коллекция будет разрушена или все ее элементы будут удалены.

Использование класса ReferenceWrapper

Если существует гарантия, что элементы, на которые мы ссылаемся, существуют, пока существует контейнер, то можно применить другой подход: использовать класс `reference_wrapper<>` (см. раздел 5.4.3).

Например, можно написать следующий код с помощью класса `Item`, введенного в предыдущем примере:

```
std::vector<std::reference_wrapper<Item>> books; // элементы – это ссылки

Item f("Faust",12.99);
books.push_back(f);                    // вставляем книгу по ссылке

// выводим книги на экран
for (const auto& book : books) {
     std::cout << book.get().getName() << ": "
               << book.get().getPrice() << std::endl;
}

f.setPrice(9.99);                      // изменяем книгу за пределами контейнера
std::cout << books[0].get().getPrice() << std::endl; // выводим на экран цену
                                                     // первой книги

// выводим на экран книги, используя тип элементов (функция get() не нужна)
for (const Item& book : books) {
   std::cout << book.getName() << ": " << book.getPrice() << std::endl;
}
```

Преимущество этого кода заключается в том, что здесь не требуется синтаксис указателей. Однако этот подход тоже рискованный, поскольку из текста не видно, что используются именно ссылки.

Например, что следующее объявление невозможно:

```
vector<Item&> books;
```

Кроме того, класс `reference_wrapper<>` содержит операцию преобразования в тип `T&`, так что можно объявить цикл `for` по интервалу, содержащему элементы типа `Item&`. Однако в этом случае необходим прямой вызов функции-члена `get()` для первого элемента.

Программы выводит на экран следующий результат (полный пример приведен в файле `cont/refwrap1.cpp`):

```
Faust: 12.99
9.99
Faust: 9.99
```

7.12. Когда и какой контейнер использовать

Стандартная библиотека C++ содержит разные контейнерные типы с разными возможностями. Возникает вопрос: когда и какой контейнерный тип использовать? Обзор ответов приведен в табл. 7.57. Однако эта таблица содержит несколько общих утверждений, которые могут не соответствовать реальности. Например, если вы управляете всего несколькими элементами, то сложность можно игнорировать, потому что короткая обработка количества элементов с линейной сложностью лучше, чем долгая обработка с логарифмической сложностью (на практике слово "несколько" может означать очень большое количество элементов).

В дополнение к таблице стоит запомнить несколько полезных правил.

- По умолчанию следует использовать вектор. Он имеет простейшую внутреннюю структуру данных и обеспечивает прямой доступ. Благодаря этому доступ к данным удобен и гибок, а обработка данных выполняется довольно быстро.
- Если вы часто вставляете и/или удаляете элементы в начале или в конце последовательности, используйте дек. Вы также должны использовать дек, если уменьшение объема внутренней памяти при удалении элементов контейнера играет существенную роль. Кроме того, поскольку вектор обычно использует один блок памяти для хранения своих элементов, дек может хранить больше элементов, потому что он использует несколько блоков.
- Если вы часто вставляете, удаляете или перемещаете элементы в середине контейнера, присмотритесь к спискам. Списки содержат специальные функции-члены для перемещения элементов из одного контейнера в другой за константное время. Однако из-за того, что списки не обеспечивают прямой доступ к элементам, при доступе к средним элементам может снизиться быстродействие, если вы находитесь в самом начале списка.
- Как все контейнеры, основанные на узлах, списки не портят итераторы, ссылающиеся на элементы, пока эти элементы являются частью контейнера. Векторы портят все свои итераторы, указатели и ссылки при превышении емкости, а также часть своих итераторов, указателей и ссылок при вставках и удалении. Деки портят итераторы, указатели и ссылки при изменении их размеров.
- Если вам нужен контейнер, обрабатывающий исключения по принципу "все или ничего", следует использовать либо список (без операций присваивания и вызова функции-члена `sort()` и, если сравнение исключений может генерировать исключение, без вызова функций-членов `merge()`, `remove()`, `remove_if()` и `unique()`; см. раздел 7.5.3), либо ассоциативный/неупорядоченный контейнер (без вызова операций вставки одновременно нескольких элементов и, если критерий сравнения может генерировать исключение при выполнении присваивания и копирования, без вызова функций-членов `swap()` или `erase()`). Обсуждение обработки исключений в библиотеке STL см. в разделе 6.12.2.

Таблица 7.57. Обзор возможностей контейнеров

	Массив	Вектор	Дек	Список	Последовательный список	Ассоциативные контейнеры	Неупорядоченные контейнеры
Доступен, начиная с	TR1	C++98	C++98	C++98	C++11	C++98	TR1
Типичная внутренняя структура данных	Статический массив	Динамический массив	Массив массивов	Двусвязный список	Односвязный список	Бинарное дерево	Хеш-таблица
Тип элементов	Значение	Значение	Значение	Значение	Значение	Множество: значение Отображение: ключ/значение	Множество: значение Отображение: ключ–значение
Разрешены ли дубликаты?	Да	Да	Да	Да	Да	Только мультимножество или мультиотображение	Только мультимножество или мультиотображение
Категория итераторов	Произвольного доступа	Произвольного доступа	Произвольного доступа	Двунаправленный	Однонаправленный	Двунаправленный (элемент/константный ключ)	Однонаправленный (элемент/константный ключ)
Возрастающий/уменьшающийся	Никогда	С одного конца	С обоих концов	Везде	Везде	Везде	Везде
Разрешен ли произвольный доступ?	Да	Да	Да	Нет	Нет	Нет	Почти
Поиск элементов	Медленно	Медленно	Медленно	Очень медленно	Очень медленно	Быстро	Очень быстро
Портит ли итераторы вставка/удаление?	—	При перераспределении памяти	Всегда	Никогда	Никогда	Никогда	При повторном хешировании
Портит ли вставка/удаление ссылки и указатели?	—	При перераспределении памяти	Всегда	Никогда	Никогда	Никогда	Никогда
Допускается ли резервирование памяти?	—	Да	Нет	—	—	—	Да (сегменты)
Освобождается ли память после удаления элементов?	—	Только функцией `shrink_to_fit()`	Иногда	Всегда	Всегда	Всегда	Иногда
Безопасность транзакций (принцип "все или ничего")	Нет	Заталкивание/выталкивание в конце	Заталкивание/выталкивание в начале и в конце	Все вставки и все удаления	Все вставки и все удаления	Вставки отдельных элементов и все удаления, если критерий сравнение не генерирует исключения	Вставки отдельных элементов и все удаления, если хеш-функция и критерий сравнение не генерируют исключения

- Если вам часто приходится искать элементы по определенному критерию, используйте неупорядоченное множество или мультимножество, выполняющее хеширование в соответствии с этим критерием. Однако хешированные контейнеры не упорядочены, поэтому, если вам требуется учитывать порядок элементов, используйте множества или мультимножества, упорядочивающие элементы по критерию поиска.
- Для обработки пар "ключ–значение" используйте неупорядоченное (мульти)отображение или, если порядок элементов имеет значение, обычное (мульти)отображение.
- Если вам нужен ассоциативный массив, используйте неупорядоченное отображение, или, если порядок элементов имеет значение, обычное отображение.
- Если вам нужен словарь, используйте неупорядоченное мультиотображение, или, если порядок элементов имеет значение, обычное мультиотображение.

Проблема заключается в том, что нелегко выбрать способ сортировки объектов по двум разным критериям. Например, возможно, вы должны хранить элементы в порядке, заданном пользователем, а поиск выполнять по другому критерию. Как и в базах данных, вам необходим быстрый доступ к двум или нескольким критериям. В данном случае вы, возможно, будете использовать два множества или два отображения, хранящие одинаковые объекты, но с разными критериями сортировки. Однако хранение объектов в двух коллекциях — это отдельная тема, которая рассматривается в разделе 7.11.

Автоматическая сортировка ассоциативных контейнеров не означает, что эти контейнеры работают лучше. Это объясняется тем, что ассоциативные контейнеры сортируют элементы каждый раз при выставке нового элемента. Часто более высокого быстродействия можно достичь с помощью последовательного контейнера и сортировки всех элементов после всех вставок, применяя несколько алгоритмов сортировки (см. раздел 11.2.2).

Следующие два простых примера сортируют все строки, считанные из стандартного потока ввода, и выводят их на экран без дубликатов, используя два разных контейнера.

1. Использование класса **set**.

```
// cont/sortset.cpp

#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
#include <set>
using namespace std;

int main()
{
   // создаем множество строк
   // - инициализируем его всеми словами из стандартного потока ввода
   set<string> coll((istream_iterator<string>(cin)),
   istream_iterator<string>());

   // выводим на экран все элементы
   copy (coll.cbegin(), coll.cend(),
         ostream_iterator<string>(cout, "\n"));
}
```

2. Использование класса **vector**.

```
// cont/sortvec.cpp

#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>
using namespace std;

int main()
{
   // создаем вектор строк
   // - инициализируем всеми словами из стандартного потока ввода
   vector<string> coll((istream_iterator<string>(cin)),
                       istream_iterator<string>());
   // сортируем элементы
   sort (coll.begin(), coll.end());

   // выводим на экран все элементы, игнорируя последующие дубликаты
   unique_copy (coll.cbegin(), coll.cend(),
                ostream_iterator<string>(cout, "\n"));
}
```

Когда я испытал эти программы на примерно 350 000 строках в одной и той же системе, вектор оказался примерно на 10% быстрее. Вставка вызова функции-члена `reserve()` повысила быстродействие вектора еще на 5%. Разрешение дубликатов, — используя класс `multiset`, а не `set`, и вызывая функцию-член `copy()`, а не `unique_copy()`, соответственно — резко изменило ситуацию: вектор стал быстрее мультимножества уже на 40%. Однако в другой системе вектор оказался на 50% медленнее. Эти измерения не являются репрезентативными, но они демонстрируют, что целесообразно испытывать разные способы обработки элементов.

На практике предсказать, какой контейнер окажется лучшим, часто бывает трудно. Большое преимущество библиотеки STL заключается в том, что мы можем испытывать разные версии без больших усилий. Основная работа —реализация разных структур данных и алгоритмов — уже сделана. Программисту остается лишь объединить их оптимальным образом.

Глава 8

Детальное описание контейнеров STL

В главе подробно обсуждаются все операции, предусмотренные в контейнерах STL. Классы и их члены сгруппированы по функциональному признаку. Для каждого типа и операции в главе приведены их сигнатуры, поведение и контейнерные типы, которые их предоставляют. Возможными контейнерными типами являются массив, вектор, дек, список, последовательный список, множество, мультимножество, отображение, мультиотображение, неупорядоченное множество, неупорядоченное мультимножество, неупорядоченное отображение, неупорядоченное мультиотображение и строка. В следующих разделах слово "контейнер" означает контейнерный тип, содержащий соответствующий член.

8.1. Определения типов

контейнер::**value_type**

- Тип элементов.
- Для (неупорядоченных) множеств и мультимножеств является константным.
- Для (неупорядоченных) отображений и мультиотображений является классом `pair <const` *тип-ключа*`,` *отображаемый-тип*`>`.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер::**reference**

- Тип ссылок на элементы.
- Как правило, *контейнер*`::value_type&`.
- В классе `vector<bool>` является вспомогательным классом (см. раздел 7.3.6).
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер::**const_reference**

- Тип ссылок на элементы только для чтения.
- Как правило, `const` *контейнер*`::value_type&`.

- В классе `vector<bool>` является типом `bool`.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер::**iterator**

- Тип итераторов.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер::**const_iterator**

- Тип итераторов только для чтения.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер::**reverse_iterator**

- Тип обратных итераторов.
- Предусмотрен в массиве, векторе, деке, списке, множестве, мультимножестве, отображении, мультиотображении и строке.
- *контейнер*::**const_reverse_iterator**
- Тип обратных итераторов только для чтения.
- Предусмотрен в массиве, векторе, деке, списке, множестве, мультимножестве, отображении, мультиотображении и строке.

контейнер::**pointer**

- Тип указателей на элементы.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер::**const_pointer**

- Тип указателей на элементы только для чтения.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер::**size_type**

- Целочисленный тип для размера без знака.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном

множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер: : **difference_type**

- Целочисленный тип для размера со знаком.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер: : **key_type**

- Тип ключей элементов ассоциативных и неупорядоченных контейнеров.
- В (неупорядоченных) множествах и мультимножествах эквивалентен типу `value_type`.
- Предусмотрен в множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении.

контейнер: : **mapped_type**

- Тип значения элементов ассоциативных и неупорядоченных контейнеров.
- Предусмотрен в отображении, мультиотображении, неупорядоченном отображении, неупорядоченном мультиотображении.

контейнер: : **key_compare**

- Тип критерия сравнения в ассоциативном контейнере.
- Предусмотрен в множестве, мультимножестве, отображении и мультиотображении.

контейнер: : **value_compare**

- Тип критерия сравнения для типа элементов.
- В множествах и мультимножествах эквивалентен `key_compare`.
- В отображения и мультиотображениях является вспомогательным классом для критерия сравнения, предназначенного для сравнения ключей двух элементов.
- Предусмотрен в множестве, мультимножестве, отображении и мультиотображении.

контейнер: : **hasher**

- Тип функции хеширования в неупорядоченных контейнерах.
- Предусмотрен в неупорядоченном множествах, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

контейнер: : **key_equal**

- Тип предиката эквивалентности в неупорядоченных контейнерах.
- Предусмотрен в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

контейнер::**local_iterator**

- Тип сегментных итераторов в неупорядоченных контейнерах.
- Доступен, начиная со стандарта C++11.
- Предусмотрен в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

контейнер::**const_local_iterator**

- Тип сегментных итераторов только для чтения в неупорядоченных контейнерах.
- Доступен, начиная со стандарта C++11.
- Предусмотрен в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

Кроме того, тип `allocator_type` предусмотрен для всех контейнеров, за исключением массивов (см. раздел 8.10.1).

8.2. Операции создания, копирования и удаления

Контейнеры имеют конструкторы и деструкторы, перечисленные ниже. Кроме того, большинство конструкторов позволяют передавать модель памяти в качестве дополнительного аргумента, указанного в разделе 8.10. Обсуждение вопросов, связанных с инициализацией контейнеров, приведено в разделе 7.1.2.

контейнер::*контейнер* ()

- Конструктор по умолчанию.
- Создает новый пустой контейнер.
- В массивах эта операция определена неявно и создает непустой контейнер, в котором элементы могут иметь неопределенные значения (см. раздел 7.2.1).
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`explicit` *контейнер*::*контейнер* (`const CompFunc&` *cmpPred*)

- Создает новый пустой контейнер с предикатом *cmpPred* в качестве критерия сортировки (см. примеры в разделах 7.7.5 и 7.8.6).
- Критерий сортировки должен определять *строгое слабое упорядочение* (см. раздел 7.7).
- Предусмотрен в множестве, мультимножестве, отображении и мультиотображении.

`explicit` *контейнер*::*контейнер* (`size_type` *bnum*)
`explicit` *контейнер*::*контейнер* (`size_type` *bnum*, `const Hasher&` *hasher*)
`explicit` *контейнер*::*контейнер* (`size_type` *bnum*, `const Hasher&` *hasher*, `const KeyEqual&` *eqPred*)

- Создают новый пустой контейнер, имеющий не менее *bnum* сегментов, функцию хеширования *hasher* и критерий эквивалентности *eqPred*.

- Если критерий эквивалентности *eqPred* не передается, используется критерий эквивалентности, предусмотренный по умолчанию для контейнерного типа (см. раздел 7.9.2).
- Если функция хеширования *hasher* не передается, используется функция хеширования, предусмотренная по умолчанию для контейнерного типа (см. раздел 7.9.2).
- Предусмотрены в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

контейнер::*контейнер* (*список-инициализации*)

- Создает новый контейнер, инициализированный элементами *списка инициализации*.
- Для массивов эта операция определена неявно (см. раздел 7.2.1).
- Доступен начиная со стандарта C++11.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер::*контейнер* (*список-инициализации*, `const CompFunc&` *cmpPred*)

- Создает контейнер, имеющий критерий сортировки *cmpPred* и инициализированный *списком инициализации*.
- Критерий сортировки должен определять *строгое слабое упорядочение* (см. раздел 7.7).
- Доступен начиная со стандарта C++11.
- Предусмотрен в множестве, мультимножестве, отображении и мультиотображении.

контейнер::*контейнер* (*список-инициализации*, `size_type` *bnum*)
контейнер::*контейнер* (*список-инициализации*, `size_type` *bnum*, `const Hasher&` *hasher*)
контейнер::*контейнер* (*список-инициализации*, `size_type` *bnum*, `const Hasher&` *hasher*, `const KeyEqual&` *eqPred*)

- Создает контейнер, имеющий не менее *bnum* сегментов, функцию хеширования *hasher*, критерий эквивалентности *eqPred* и инициализированный элементами *списка инициализации*.
- Если критерий эквивалентности *eqPred* не передается, используется критерий эквивалентности, предусмотренный для контейнерного типа по умолчанию (см. раздел 7.9.2).
- Если функция хеширования *hasher* не передается, используется функция хеширования, предусмотренная для контейнерного типа по умолчанию (см. раздел 7.9.2).
- Предусмотрены в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

контейнер::*контейнер* (`const` *контейнер*`&` *c*)

- Копирующий конструктор.
- Создает новый контейнер как копию существующего контейнера *c*.

- Вызывает копирующий конструктор каждого элемента контейнера *c*.
- Для массивов эта операция определена неявно.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер::*контейнер* (*контейнер*&& *c*)

- Перемещающий конструктор.
- Создает новый контейнер, инициализированный элементами существующего контейнера *c*.
- После вызова перемещающего конструктора контейнер *c* остается корректным, но имеет неопределенное значение.
- Для массивов эта операция определена неявно.
- Доступна, начиная со стандарта C++11.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

explicit *контейнер*::*контейнер* (size_type *num*)

- Создает контейнер, состоящий из *num* элементов.
- Элементы создаются своими конструкторами, заданными по умолчанию.
- Предусмотрен в векторе, деке, списке и последовательном списке.

контейнер::*контейнер* (size_type *num*, const T& *value*)

- Создает контейнер, состоящий из *num* элементов.
- Элементы создаются как копии аргумента *value*.
- T — тип элементов контейнера.
- Для строк аргумент *value* не передается по ссылке.
- Предусмотрен в векторе, деке, списке, последовательном списке и строке.

контейнер::*контейнер* (InputIterator *beg*, InputIterator *end*)

- Создает контейнер, инициализированный всеми элементами диапазона [*beg*,*end*).
- Эта функция-член является шаблонной (см. раздел 3.2). Следовательно, элементы диапазона-источника могут иметь любой тип, который может быть преобразован в тип элементов контейнера.
- Примеры и обсуждения проблем, возникающих при работе с этой функцией-членом, см. в разделе 7.1.2.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер::*контейнер* (InputIterator *beg*, InputIterator *end*,
const CompFunc& *cmpPred*)

- Создает контейнер, имеющий критерий сортировки *cmpPred* и инициализированный всеми элементами диапазона [*beg*,*end*).
- Эта функция-член является шаблонной (см. раздел 3.2). Следовательно, элементы диапазона-источника могут иметь любой тип, который может быть преобразован в тип элементов контейнера.
- Критерий сортировки должен определять *строгое слабое упорядочение* (см. раздел 7.7).
- Предусмотрен в множестве, мультимножестве, отображении, мультиотображении.

контейнер::*контейнер* (InputIterator *beg*, InputIterator *end*,
size_type *bnum*)
контейнер::*контейнер* (InputIterator *beg*, InputIterator *end*,
size_type *bnum*, const Hasher& *hasher*)
контейнер::*контейнер* (InputIterator *beg*, InputIterator *end*,
size_type *bnum*, const Hasher& *hasher*, const KeyEqual& *eqPred*)

- Создает контейнер, имеющий не менее *bnum* сегментов, функцию хеширования *hasher*, критерий эквивалентности *eqPred* и инициализированный всеми элементами диапазона [*beg*,*end*).
- Если критерий эквивалентности *eqPred* не передается, используется критерий эквивалентности, предусмотренный по умолчанию для данного контейнерного типа (см. раздел 7.9.2).
- Если функция хеширования *hasher* не передается, используется функция хеширования, предусмотренная по умолчанию для данного контейнерного типа (см. раздел 7.9.2).
- Предусмотрены в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

контейнер::~*контейнер* ()

- Деструктор.
- Удаляет все элементы и освобождает память.
- Вызывает деструктор каждого элемента.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

8.3. Немодифицирующие операции

8.3.1. Операции над размером

bool *контейнер*::**empty** () const

- Возвращает признак того, что контейнер пуст (не содержит элементов).
- Эквивалент выражения begin()==end(), но может работать быстрее.

- Сложность: константная.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`size_type` *контейнер*`::`**size** `() const`

- Возвращает текущее количество элементов.
- Для проверки, пуст ли контейнер (не содержит элементов), необходимо использовать функцию-член `empty()`, которая может работать быстрее.
- Сложность: константная.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`size_type` *контейнер*`::`**max_size** `() const`

- Возвращает максимальное количество элементов, которые может содержать контейнер.
- Это техническое значение, которое может зависеть от модели памяти, используемой в контейнере. В частности, поскольку векторы всегда занимают один сегмент памяти, эта величина для вектора может оказаться меньше, чем для других контейнеров.
- Сложность: константная.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

8.3.2. Операции сравнения

`bool` **operator ==** `(const` *контейнер*`& c1, const` *контейнер*`& c2)`
`bool` **operator !=** `(const` *контейнер*`& c1, const` *контейнер*`& c2)`

- Возвращает признак того, что два контейнера (не) равны.
- Два контейнера считаются равными, если они содержат одинаковое количество элементов и состоят из одних и тех же элементов (т.е. для всех сравнений соответствующих элементов операция == возвращает значение `true`). За исключением неупорядоченных контейнеров, одинаковые элементы хранятся в одном и том же порядке.
- Сложность: как правило, линейная. Для неупорядоченных контейнеров в наихудшем случае является квадратичной.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

```
bool operator < (const контейнер& c1, const контейнер& c2)
bool operator <= (const контейнер& c1, const контейнер& c2)
bool operator > (const контейнер& c1, const контейнер& c2)
bool operator >= (const контейнер& c1, const контейнер& c2)
```

- Возвращают результат сравнения двух контейнеров одинакового типа.
- Для проверки того, является ли один контейнер меньше другого, контейнеры сравниваются в лексикографическом порядке (описание алгоритма `lexicographical_compare()` приведено в разделе 11.5.4).
- Сложность: линейная.
- Предусмотрен в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, строке.

8.3.3. Немодифицирующие операции над ассоциативными и неупорядоченными контейнерами

Функции-члены, перечисленные в этом разделе, представляют собой специальные реализации соответствующих алгоритмов STL, рассмотренных в разделах 11.5 и 11.9. Эти функции-члены обеспечивают более высокую производительность, потому что они используют тот факт, что элементы ассоциативных контейнеров упорядочиваются и фактически гарантируют логарифмическую, а не линейную сложность операций. Например, поиск элемента среди 1000 элементов в среднем требует не более 10 сравнений (см. раздел 2.2). Если используется хорошая хеш-функция, то функции-члены могут иметь даже константную сложность.

```
size_type контейнер::count (const T& value) const
```

- Возвращает количество элементов, эквивалентных аргументу *value*.
- Представляет собой специальную версию алгоритма `count()` из раздела 11.5.1.
- `T` — тип упорядоченных значений:
  - для множеств и мультимножеств является типом элементов;
  - для отображения и мультиотображений является типом ключей.
- Сложность: логарифмическая для ассоциативных массивов и константная для неупорядоченных контейнеров, снабженных хорошей хеш-функцией.
- Предусмотрен в множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении.

```
iterator контейнер::find (const T& value)
const_iterator контейнер::find (const T& value) const
```

- Возвращают позицию первого элемента, имеющего значение, эквивалентное аргументу *value*.
- Если элемент не найден, возвращают итератор `end()`.
- Представляют собой специальные версии алгоритма `find()`, рассмотренного в разделе 11.5.3.

- `T` — тип упорядоченных значений:
  - для множеств и мультимножеств является типом элементов;
  - для отображений и мультиотображений является типом ключей.
- Сложность: логарифмическая для ассоциативных массивов и константная для неупорядоченных контейнеров, снабженных хорошей хеш-функцией.
- Предусмотрен в множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении.

`iterator` *контейнер*`::`**lower_bound** `(const T&` *value*`)`
`const_iterator` *контейнер*`::`**lower_bound** `(const T&` *value*`) const`

- Возвращают первую позицию, в которую можно вставить копию аргумента *value* в соответствии с критерием сортировки.
- Если элемент не найден, возвращают итератор `end()`.
- Возвращаемое значение представляет собой позицию первого элемента, равного или большего аргумента *value*, в соответствии с критерием сортировки (возможно `end()`).
- Представляют собой специальные версии алгоритма `lower_bound()`, рассмотренного в разделе 11.10.1.
- `T` — тип упорядоченных значений:
  - для множеств и мультимножеств является типом элементов;
  - для отображения и мультиотображений является типом ключей.
- Сложность: логарифмическая.
- Предусмотрены в множестве, мультимножестве, отображении и мультиотображении.

`iterator` *контейнер*`::`**upper_bound** `(const T&` *value*`)`
`const_iterator` *контейнер*`::`**upper_bound** `(const T&` *value*`) const`

- Возвращают последнюю позицию, в которую можно вставить копию аргумента *value* в соответствии с критерием сортировки.
- Если элемент не найден, возвращают итератор `end()`.
- Возвращаемое значение представляет собой позицию первого элемента, большего аргумента *value*, в соответствии с критерием сортировки (возможно, `end()`).
- Представляют собой специальные версии алгоритма `upper_bound()`, рассмотренного в разделе 11.10.1.
- `T` — тип упорядоченных значений:
  - для множеств и мультимножеств является типом элементов;
  - для отображений и мультиотображений является типом ключей.
- Сложность: логарифмическая.
- Предусмотрены в множестве, мультимножестве, отображении и мультиотображении.

`pair<iterator,iterator>` *контейнер*`::`**equal_range** `(const T&` *value*`)`
`pair<const_iterator,const_iterator>`
контейнер`::`**equal_range** `(const T&` *value*`) const`

- Возвращают первую и последнюю позиции, в которые можно вставить копию аргумента *value*, в соответствии с критерием сортировки.
- Возвращаемое значение представляет собой диапазон элементов, равных аргументу *value*.
- Эквивалент вызова `make_pair(lower_bound(`*value*`),upper_bound(`*value*`))`.
- Представляют собой специальные версии алгоритма `equal_range()`, рассмотренного в разделе 11.10.1.
- `T` — тип упорядоченных значений:
  - для множеств и мультимножеств является типом элементов;
  - для отображений и мультиотображений является типом ключей.
- Сложность: логарифмическая для ассоциативных массивов и константная для неупорядоченных контейнеров, снабженных хорошей хеш-функцией.
- Предусмотрен в множестве, мультимножестве, отображении и мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

8.4. Присваивание

контейнер`&` *контейнер*`::`**operator** `=` `(const` *контейнер*`&` *c*`)`

- Копирующая операция присваивания.
- Присваивает все элементы контейнера *c*; т.е. заменяет существующие элементы копиями элементов контейнера *c*.
- Эта операция может вызывать операцию присваивания для всех элементов, которые должны быть заменены, копирующий конструктор для добавляемых элементов и деструктор для удаляемых элементов.
- Предусмотрена в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер`&` *контейнер*`::`**operator** `=` `(`*контейнер*`&&` *c*`)`

- Перемещающая операция присваивания.
- Перемещает все элементы контейнера *c* в контейнер `*this`; т.е. заменяет существующие элементы копиями элементов контейнера *c*.
- После ее выполнения контейнер `c` остается корректным, но имеет неопределенное значение.
- Доступен начиная со стандарта C++11.

- Предусмотрена в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер`&` *контейнер*`::` **operator** `=` (*список-инициализации*)

- Присваивает все элементы из *списка инициализации*, т.е. заменяет все существующие элементы копиями передаваемых элементов.
- Эта операция может вызывать операцию присваивания для всех элементов, которые должны быть заменены, копирующий конструктор для добавляемых элементов и деструктор для удаляемых элементов.
- Доступен начиная со стандарта C++11.
- Предусмотрена в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`void` *контейнер*`::` **assign** (*список-инициализации*)

- Присваивает все элементы из *списка инициализации*, т.е. заменяет все существующие элементы копиями передаваемых элементов.
- Доступна, начиная со стандарта C++11.
- Предусмотрена в векторе, деке, списке, последовательном списке и строке.

`void` *массив*`::` **fill** (`const T&` *value*)

- Присваивает аргумент *value* всем элементам, т.е. заменяет все существующие элементы копиями аргумента *value*.
- Доступна начиная со стандарта C++11.
- Предусмотрена в массиве.

`void` *контейнер*`::` **assign** (`size_type` *num*`,` `const T&` *value*)

- Присваивает *num* вхождений аргумента *value*, т.е. заменяет все существующие элементы *num* копиями аргумента *value*.
- `T` должен быть типом элемента.
- Предусмотрена в векторе, деке, списке, последовательном списке и строке.

`void` *контейнер*`::` **assign** (`InputIterator` *beg*`,` `InputIterator` *end*)

- Присваивает все элементы диапазона [*beg*,*end*), т.е. заменяет все существующие элементы копиями элементов диапазона [*beg*,*end*).
- Эта функция-член является шаблонной (см. раздел 3.2). Следовательно, элементы диапазона-источника могут иметь любой тип, который может быть преобразован в тип элементов контейнера.
- Предусмотрена в векторе, деке, списке, последовательном списке и строке.

`void` *контейнер*`::`**swap** `(`*контейнер*`&` *c*`)`
`void` **swap** `(`*контейнер*`&` *c1*`,` *контейнер*`&` *c2*`)`

- Обменивает содержимое текущего контейнера и контейнера *c* или содержимое контейнеров *c1* и *c2* соответственно.
- Обе функции выполняют обмен
  - элементов контейнеров;
  - их критериев сортировки, предикатов эквивалентности и хеш-функций, если таковые существуют.
- Ссылки, указатели и итераторы, установленные на элементы, меняют свои контейнеры, потому что после обмена они ссылаются на прежние элементы.
- Массивы не могут просто обменять внутренние указатели. Следовательно, функция `swap()` имеет линейную сложность, а итераторы и ссылки после обмена относятся к тому же самому контейнеру, но с другими элементами.
- Для ассоциативных контейнеров функция может генерировать исключение, только если копирование и присваивание критерия сравнения могут генерировать исключение. Для неупорядоченных контейнеров функция может генерировать исключение, только если предикат эквивалентности или хеш-функция могут генерировать исключение. Для всех остальных контейнеров функция не может генерировать исключение.
- Сложность: как правило, константная. Для массивов сложность является линейной.
- Если присваиваемый объект далее не нужен, то из-за невысокой сложности функции `swap()` всегда следует отдавать предпочтение ей, а не копирующему присваиванию (см. раздел 7.1.2).
- Предусмотрена в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

8.5. Прямой доступ к элементам

`reference` *контейнер*`::`**at** `(size_type` *idx*`)`
`const_reference` *контейнер*`::`**at** `(size_type` *idx*`) const`

- Возвращают элемент с индексом *idx* (первый элемент имеет индекс `0`).
- Передача некорректного индекса (меньшего `0` или равного `size()` или большего `size()`) вызывает генерацию исключения `out_of_range`.
- Возвращаемая ссылка может стать некорректной из-за последующих модификаций или перераспределения памяти.
- Если вы уверены, что индекс корректен, можно использовать операцию `[]`, которая работает быстрее.
- Предусмотрены в массиве, векторе, деке и строке.

`T& `*отображение*`::`**at** `(const key_type& `*key*`)`
`const T& `*отображение*`::`**at** `(const key_type& `*key*`) const`

- Возвращают значение, соответствующее ключу *key* в отображении.
- Если в отображении нет элемента, ключ которого совпадает с аргументом *key*, генерируют исключение `out_of_range`.
- Доступны начиная со стандарта C++11.
- Предусмотрены в отображении и неупорядоченном отображении.

`reference `*контейнер*`::`**operator[]** `(size_type `*idx*`)`
`const_reference `*контейнер*`::`**operator[]** `(size_type `*idx*`) const`

- Возвращают элемент с индексом *idx* (первый элемент имеет индекс `0`).
- Передача неправильного индекса (меньшего `0` или равного `size()` или большего `size()`) приводит к неопределенному поведению. Следовательно, вызывающая сторона должна гарантировать, что индекс является корректным; в противном случае следует использовать функцию-член `at()`.
- Возвращаемая ссылка может стать некорректной из-за последующих модификаций или перераспределения памяти.
- Предусмотрены в массиве, векторе, деке и строке.

`T& `*отображение*`::`**operator[]** `(const key_type& `*key*`)`
`T& `*отображение*`::`**operator[]** `(key_type&& `*key*`)`

- Операции `[]` для ассоциативных массивов.
- Возвращают значение, соответствующее ключу *key* в отображении.
- Если в отображении нет элемента, ключ которого совпадает с аргументом *key*, эти операции автоматически *создают* новый элемент с данным ключом (копируемым или перемещаемым) и значением, инициализированным конструктором, предусмотренным по умолчанию для данного типа значения. Таким образом, индекс никогда не может быть некорректным (возможно только неправильное поведение). Подробнее см. в разделах 6.2.4 и 7.8.3.
- Во второй версии состояние аргумента *key* после выполнения операции становится неопределенным (эта форма реализует семантику перемещения для случая, когда ключа еще не существует).
- Первая версия эквивалентна выражению

`(*((insert(make_pair(`*key*`,T()))).first)).second`

- Вторая версия доступна начиная со стандарта C++11.
- Предусмотрены в отображении и мультиотображении.

`reference `*контейнер*`::`**front** `()`
`const_reference `*контейнер*`::`**front** `() const`

- Обе версии возвращают первый элемент (элемент с индексом `0`).
- Вызывающая сторона должна гарантировать, что контейнер содержит хотя бы один элемент (`size()>0`); в противном случае последствия непредсказуемы.

- Для строк обе операции доступны начиная со стандарта C++11.
- Предусмотрены в массиве, векторе, деке, списке, последовательном списке и строке.

```
reference контейнер::back ()
const_reference контейнер::back () const
```

- Обе функции возвращают последний элемент (элемент с индексом `size()-1`).
- Вызывающая сторона должна гарантировать, что контейнер содержит хотя бы один элемент (`size()>0`); в противном случае последствия непредсказуемы.
- Для строк обе операции доступны начиная со стандарта C++11.
- Предусмотрены в массиве, векторе, деке, списке и строке.

```
T* контейнер::data ()
const T* контейнер::data () const
```

- Обе функции возвращают обычный массив в стиле языка C со всеми элементами (т.е. указатель на первый элемент массива).
- Обе функции предназначены для передачи элементов массива интерфейсам в стиле языка C.
- Для строк доступна только вторая версия.
- Для массивов и векторов доступны начиная со стандарта C++11.
- Предусмотрены в массиве, векторе и строке.

8.6. Операции генерации итераторов

Следующие функции-члены возвращают итераторы для обхода элементов контейнера. В табл. 8.1 приведены категории итераторов (см. раздел 9.2) в соответствии с разными типами контейнеров.

Таблица 8.1. Категории итераторов, предусмотренных для контейнерных типов

Контейнер	Категория итератора
Массив	Произвольного доступа
Вектор	Произвольного доступа
Дек	Произвольного доступа
Список	Двунаправленный
Последовательный список	Однонаправленный
Множество	Двунаправленный; элемент — константа
Мультимножество	Двунаправленный; элемент — константа
Отображение	Двунаправленный; ключ — константа

Окончание табл. 8.1

Контейнер	Категория итератора
Мультиотображение	Двунаправленный; ключ — константа
Неупорядоченное множество	Однонаправленный; элемент — константа
Неупорядоченное мультимножество	Однонаправленный; элемент — константа
Неупорядоченное отображение	Однонаправленный; ключ — константа
Неупорядоченное мультиотображение	Однонаправленный; ключ — константа
Строка	Произвольного доступа

```
iterator контейнер::begin ()
const_iterator контейнер::begin () const
const_iterator контейнер::cbegin () const
```

- Возвращает итератор, установленный в начало контейнера (позицию первого элемента).
- Если контейнер пуст, вызов функции эквивалентен вызовам *контейнер*`::end()` или *контейнер*`::cend()` соответственно.
- В неупорядоченных контейнерах есть функции-члены `begin()` и `cbegin()` с числовым аргументом для обеспечения сегментного интерфейса (см. раздел 8.9.3).
- Функция `cbegin()` доступна начиная со стандарта C++11.
- Предусмотрена в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

```
iterator контейнер::end ()
const_iterator контейнер::end () const
const_iterator контейнер::cend () const
```

- Возвращает итератор, установленный в конец контейнера (позицию после последнего элемента).
- Если контейнер пуст, вызов функции эквивалентен вызовам *контейнер*`::begin()` или *контейнер*`::cbegin()` соответственно.
- В неупорядоченных контейнерах есть функции-члены `begin()` и `cbegin()` с числовым аргументом для обеспечения сегментного интерфейса (см. раздел 8.9.3).
- Функция `cend()` доступна начиная со стандарта C++11.
- Предусмотрена в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`reverse_iterator` *контейнер*`::`**rbegin** `()`
`const_reverse_iterator` *контейнер*`::`**rbegin** `() const`
`const_reverse_iterator` *контейнер*`::`**crbegin** `() const`

- Возвращает обратный итератор, установленный в начало контейнера при обратном обходе (позицию последнего элемента).
- Если контейнер пуст, вызов функции эквивалентен вызовам *контейнер*`::rend()` или *контейнер*`::crend()` соответственно.
- Подробное описание обратных итераторов приведено в разделе 9.4.1.
- Функция `crbegin()` доступна начиная со стандарта C++11.
- Предусмотрена в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении и строке.

`reverse_iterator` *контейнер*`::`**rend** `()`
`const_reverse_iterator` *контейнер*`::`**rend** `() const`
`const_reverse_iterator` *контейнер*`::`**crend** `() const`

- Возвращает обратный итератор, установленный в конец контейнера при обратном обходе (позицию перед первым элементом).
- Если контейнер пуст, вызов функции эквивалентен вызовам *контейнер*`::rbegin()` или *контейнер*`::crbegin()` соответственно.
- Подробное описание обратных итераторов приведено в разделе 9.4.1.
- Функция `crend()` доступна начиная со стандарта C++11.
- Предусмотрена в массиве, векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении и строке.

8.7. Вставка и удаление элементов

8.7.1. Вставка отдельных элементов

`iterator` *контейнер*`::`**insert** `(const T&` *value*`)`
`iterator` *контейнер*`::`**insert** `(T&&` *value*`)`
`pair<iterator,bool>` *контейнер*`::`**insert** `(const T&` *value*`)`
`pair<iterator,bool>` *контейнер*`::`**insert** `(T&&` *value*`)`

- Вставляют аргумент *value* в ассоциативный или неупорядоченный контейнер.
- Первая и третья версии копируют аргумент *value*.
- Вторая и четвертая версии перемещают аргумент *value* в контейнер, так что после выполнения операций значение аргумента *value* становится неопределенным.
- Контейнеры, допускающие дубликаты (неупорядоченные), мультимножества и мультиотображения, имеют первую и вторую сигнатуры. Они возвращают позицию нового элемента. Начиная со стандарта C++11 вновь вставленные элементы гарантированно размещаются после существующих эквивалентных значений.

- Контейнеры, не допускающие дубликатов (неупорядоченные), множества и отображения, имеют третью и четвертую сигнатуры. Если вставка элемента невозможна из-за того, что в контейнере уже существует элемент с таким же значением или ключом, они возвращают позицию существующего элемента и значение `false`. Если вставка возможна, они возвращают позицию нового элемента и значение `true`.
- `T` — тип элементов контейнера. Следовательно, для (неупорядоченных) отображений и мультиотображений он представляет собой пару "ключ–значение".
- Для отображения, мультиотображения, неупорядоченного отображения и неупорядоченного мультиотображения соответствующая форма с семантикой перемещения является шаблонной (см. раздел 3.2). Следовательно, аргумент *value* может иметь любой тип, который можно преобразовать в тип значения (пару "ключ–значение") контейнера. Это позволяет передавать две строки так, чтобы первая из них могла быть преобразована в константную строку (ключ).
- Если хеш-функции в неупорядоченных контейнерах не генерируют исключений, функции работают по принципу "все или ничего".
- Для всех контейнеров ссылки на существующие элементы остаются корректными. Для ассоциативных контейнеров все итераторы, установленные на существующие элементы, остаются корректными. Для неупорядоченных контейнеров итераторы, установленные на существующие элементы, остаются корректными, если не выполняется повторное хеширование (если количество результирующих элементов равно или больше количества сегментов, умноженного на коэффициент максимальной нагрузки).
- Вторая и четвертая версии доступны начиная со стандарта C++11.
- Предусмотрены в множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении.

`iterator` *контейнер*`::`**emplace** (*args*)
`pair<iterator,bool>` *контейнер*`::`**emplace** (*args*)

- Вставляет новый элемент, инициализированный списком аргументов *args*, в ассоциативный или неупорядоченный массив.
- Контейнеры, допускающие дубликаты (неупорядоченные), мультимножества и мультиотображения, имеют первую сигнатуру. Они возвращают позицию нового элемента. Начиная со стандарта C++11 вновь вставленные элементы гарантированно размещаются после существующих эквивалентных значений.
- Контейнеры, не допускающие дубликатов (неупорядоченные), множества и отображения, имеют вторую сигнатуру. Если вставка элемента невозможна из-за того, что в контейнере уже существует элемент с таким же значением или ключом, они возвращают позицию существующего элемента и значение `false`. Если вставка возможна, они возвращают позицию нового элемента и значение `true`.
- Если хеш-функции в неупорядоченных контейнерах не генерируют исключений, функции работают по принципу "все или ничего".
- Для всех контейнеров ссылки на существующие элементы остаются корректными. Для ассоциативных контейнеров все итераторы, установленные на существующие элементы, остаются корректными. Для неупорядоченных контейнеров итераторы,

установленные на существующие элементы, остаются корректными, если не выполняется повторное хеширование (если количество результирующих элементов равно или больше количества сегментов, умноженного на коэффициент максимальной нагрузки).

- Для последовательных контейнеров возможна та же сигнатура, в которой первый аргумент рассматривается как позиция, в которую можно вставить новый элемент (см. раздел 8.7.1).
- Для вставки новой пары "ключ–значение" в (неупорядоченные) отображения и мультиотображения необходимо использовать конструкцию, работающую частями, в виде кортежей (см. раздел 7.8.2).
- Доступны начиная со стандарта C++11.
- Предусмотрены в множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении.

`iterator` *контейнер*`::`**`insert`** `(const_iterator` *pos*`, const T&` *value*`)`
`iterator` *контейнер*`::`**`insert`** `(const_iterator` *pos*`, T&&` *value*`)`

- Вставляет аргумент *value* в позицию итератора *pos*.
- Первая версия копирует аргумент *value*.
- Вторая версия перемещает аргумент *value* в контейнер, так что после выполнения операции значение аргумента *value* становится неопределенным.
- Возвращает позицию нового элемента.
- Если контейнер не допускает дубликатов (множества, неупорядоченные множества, отображения и неупорядоченные отображения) и уже содержит элемент, равный (ключу) *value*, операция не имеет последствий и возвращает позицию существующего элемента.
- Для ассоциативных и неупорядоченных контейнеров позиция используется только как подсказка, откуда можно начинать поиск места для вставки. Если аргумент *value* вставляется прямо в позицию *pos*, то функция имеет амортизированную константную сложность; в противном случае она имеет логарифмическую сложность.
- При работе с векторами эта операция делает итераторы и ссылки на элементы некорректными, если эти элементы предшествуют позиции *pos* и не выполняется перераспределение памяти.
- При работе с деками эта операция делает итераторы некорректными, а если позиция *pos* не совпадает с одним из концов, но некорректными становятся и все ссылки.
- `T` — тип элементов контейнера. Следовательно, для (неупорядоченных) отображений и мультиотображений он представляет собой пару "ключ–значение".
- Для отображения, мультиотображения, неупорядоченного отображения и неупорядоченного мультимножества соответствующая форма с семантикой перемещения является шаблонной (см. раздел 3.2). Следовательно, аргумент *value* может иметь любой тип, который можно преобразовать в тип значения (пару "ключ–значение") контейнера. Это позволяет передавать две строки так, чтобы первая из них могла быть преобразована в константную строку (ключ).
- При работе со строками аргумент *value* передается по значению.

- Если при работе с векторами и деками операции копирования и перемещения (конструктор и операция присваивания) элементов не генерируют исключений, то функция работает по принципу "все или ничего". Если хеш-функция в неупорядоченных контейнерах не генерирует исключений, функция работает по принципу "все или ничего". Во всех остальных стандартных контейнерах функция работает по принципу "все или ничего".
- Вторая версия доступна начиная со стандарта C++11. До появления стандарта C++11 использовался тип `iterator`, а не `const_iterator`.
- Предусмотрена в векторе, деке, списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`iterator` *контейнер*`::`**emplace** `(const_iterator` *pos*`,` *args*`)`

- Вставляет новый элемент, инициализированный списком аргументов *args*, в позицию итератора *pos*.
- Возвращает позицию нового элемента.
- При работе с векторами эта операция делает итераторы и ссылки на элементы некорректными, если эти элементы предшествуют позиции *pos* и не выполняется перераспределение памяти.
- При работе с деками эта операция делает итераторы некорректными, а если позиция *pos* не совпадает с одним из концов, но некорректными становятся и все ссылки.
- `T` — тип элементов контейнера.
- Если при работе с векторами и деками операции копирования и перемещения (конструктор и операция присваивания) элементов не генерируют исключений, то функция работает по принципу "все или ничего". Во всех остальных стандартных контейнерах функция работает по принципу "все или ничего".
- Для ассоциативных контейнеров возможна та же сигнатура, в которой первый аргумент рассматривается как первый аргумент нового элемента (см. раздел 8.7.1).
- Если хеш-функция в неупорядоченных контейнерах не генерирует исключений, функция работает по принципу "все или ничего".
- Доступна начиная со стандарта C++11.
- Предусмотрена в векторе, деке и списке.

`iterator` *контейнер*`::`**emplace_hint** `(const_iterator` *pos*`,` *args*`)`

- Вставляет новый элемент, инициализированный списком аргументов *args*, в позицию итератора *pos*.
- Возвращает позицию нового элемента.
- Если контейнер не допускает дубликатов (множества, неупорядоченные множества, отображения и неупорядоченные отображения) и уже содержит элемент, равный (ключу) *args*, операция не имеет последствий и возвращает позицию существующего элемента.

- Для ассоциативных и неупорядоченных контейнеров позиция используется только как подсказка, откуда можно начинать поиск места для вставки. Если новый элемент вставляется непосредственно в позицию *pos*, то функция имеет амортизированную константную сложность; в противном случае — логарифмическую сложность.
- `T` — тип элементов контейнера. Следовательно, для (неупорядоченных) отображений и мультиотображений он представляет собой пару "ключ–значение".
- Если хеш-функция в неупорядоченных контейнерах не генерирует исключений, функция работает по принципу "все или ничего".
- Доступна начиная со стандарта C++11.
- Предусмотрена в множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении.

`void` *контейнер*`::`**push_front** `(const T&` *value*`)`
`void` *контейнер*`::`**push_front** `(T&&` *value*`)`

- Вставляют аргумент *value* как новый первый элемент.
- Первая версия копирует аргумент *value*.
- Вторая версия, доступная начиная со стандарта C++11, перемещает аргумент *value* в контейнер, поэтому состояние аргумента *value* после выполнения операции становится неопределенным.
- `T` — тип элементов контейнера.
- Обе версии эквивалентны выражению `insert(begin(),` *value*`)`.
- Для деков эта операция делает итераторы на все элементы некорректными. Ссылки на элементы остаются корректными.
- Функции работают по принципу "все или ничего".[1]
- Предусмотрены в деке, списке и последовательном списке.

`void` *контейнер*`::`**emplace_front** `(`*args*`)`

- Вставляет новый первый элемент, инициализированный списком аргументов *args*.
- Таким образом, тип элемента должен иметь вызываемый конструктор для списка аргументов *args*.
- Для деков эта операция делает итераторы на все элементы некорректными. Ссылки на элементы остаются корректными.
- Функция работает по принципу "все или ничего".
- Доступна начиная со стандарта C++11.
- Предусмотрена в деке, списке и последовательном списке.

`void` *контейнер*`::`**push_back** `(const T&` *value*`)`
`void` *контейнер*`::`**push_back** `(T&&` *value*`)`

- Добавляют аргумент *value* как новый последний элемент.
- Первая версия копирует аргумент *value*.

[1] Для последовательных списков стандарт в настоящее время ничего об этом не говорит, что, вероятно, является его недочетом.

- Вторая версия, доступная начиная со стандарта C++11, перемещает аргумент *value* в контейнер, поэтому состояние аргумента *value* после выполнения операции становится неопределенным.
- `T` — тип элементов контейнера.
- Обе версии эквивалентны выражению `insert(end(), value)`.
- Для векторов эта операция делает итераторы и ссылки на все элементы некорректными, если происходит перераспределение памяти (новое количество элементов превосходит предыдущую емкость).
- Для деков эта операция делает итераторы на все элементы некорректными. Ссылки на элементы остаются корректными.
- Для строк аргумент *value* передается по значению.
- Эти функции работают по принципу "все или ничего" (для векторов и деков это гарантируется тем, что при перераспределении памяти используется копирующий конструктор, а перемещающий конструктор не генерирует исключений).
- Предусмотрены для вектора, дека, списка и строки.

`void` *контейнер*`::`**emplace_back** (*args*)

- Добавляет новый последний элемент, инициализированный списком аргументов *args*.
- Таким образом, тип элемента должен иметь вызываемый конструктор для списка аргументов *args*.
- Для векторов эта операция делает итераторы и ссылки на все элементы некорректными, если происходит перераспределение памяти (новое количество элементов превосходит предыдущую емкость).
- Для деков эта операция делает итераторы на все элементы некорректными. Ссылки на элементы остаются корректными.
- Эти функции работают по принципу "все или ничего" (для векторов и деков это гарантируется тем, что при перераспределении памяти используется копирующий конструктор, а перемещающий конструктор не генерирует исключений).
- Доступна начиная со стандарта C++11.
- Предусмотрена в векторе, деке и списке.

8.7.2. Вставка нескольких элементов

`void` *контейнер*`::`**insert** (*список-инициализации*)

- Вставляет копии элементов *списка инициализации* в ассоциативный контейнер.
- Для всех контейнеров ссылки на существующие элементы остаются корректными. Для ассоциативных контейнеров все итераторы, установленные на существующие элементы, остаются корректными. Для неупорядоченных контейнеров итераторы, установленные на существующие элементы, остаются корректными, если не выполняется повторное хеширование (когда количество результирующих элементов равно или больше количества сегментов, умноженного на коэффициент максимального заполнения).

- Доступна начиная со стандарта C++11.
- Предусмотрена в множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении.

`iterator` *контейнер*`::`**insert** `(const_iterator` *pos*`,` *список-инициализации*`)`

- Вставляет копии элементов *списка инициализации* в позицию итератора *pos*.
- Возвращает позицию первого вставленного элемента или итератор *pos*, если *список инициализации* пуст.
- При работе с векторами эта операция делает итераторы и ссылки на элементы некорректными, если эти элементы предшествуют позиции *pos* и не выполняется перераспределение памяти.
- При работе с деками эта операция делает итераторы некорректными, а если позиция *pos* не совпадает с одним из концов, но некорректными становятся и все ссылки.
- Для списков функция работает по принципу "все или ничего".
- Доступен начиная со стандарта C++11.
- Предусмотрена в векторе, деке, списке и строке.

`iterator` *контейнер*`::`**insert** `(const_iterator` *pos*`,`
`size_type` *num*`, const T&` *value*`)`

- Вставляет *num* копий аргумента *value* в позицию итератора *pos*.
- Возвращает позицию первого вставленного элемента или итератор *pos*, если *num*==0 (до появления стандарта C++11 не возвращала ничего).
- При работе с векторами эта операция делает итераторы и ссылки на элементы некорректными, если эти элементы предшествуют позиции *pos* и не выполняется перераспределение памяти.
- При работе с деками эта операция делает итераторы некорректными, а если позиция *pos* не совпадает с одним из концов, но некорректными становятся и все ссылки.
- `T` — тип элементов контейнера. Следовательно, для отображений и мультиотображений он представляет собой пару "ключ–значение".
- Для строк аргумент *value* передается по значению.
- Если при работе с векторами и деками операции копирования и перемещения (конструктор и операция присваивания) элементов не генерируют исключений, то функция работает по принципу "все или ничего". При работе со списками функция работает по принципу "все или ничего".
- До появления стандарта C++11 использовался тип `iterator`, а не `const_iterator`, а возвращаемое значение имело тип `void`.
- Предусмотрена в векторе, деке, списке и строке.

`void` *контейнер*`::`**insert** `(InputIterator` *beg*`, InputIterator` *end*`)`

- Вставляет в ассоциативный контейнер копии всех элементов диапазона [*beg*,*end*).
- Эта функция-член является шаблонной (см. раздел 3.2). Следовательно, элементы диапазона-источника могут иметь любой тип, который может быть преобразован в тип элементов контейнера.

- Для всех контейнеров ссылки на существующие элементы остаются корректными. Для ассоциативных контейнеров все итераторы, установленные на существующие элементы, остаются корректными. Для неупорядоченных контейнеров итераторы, установленные на существующие элементы, остаются корректными, если не выполняется повторное хеширование (если количество результирующих элементов равно или больше количества сегментов, умноженного на коэффициент максимального заполнения).
- Если хеш-функции в неупорядоченных контейнерах не генерируют исключений, функции работают по принципу "все или ничего".
- Предусмотрена в множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении.

`iterator` *контейнер*`::`**insert** `(const_iterator` *pos*`,`
`InputIterator` *beg*`, InputIterator` *end*`)`

- Вставляет в ассоциативный контейнер копии всех элементов диапазона [*beg*,*end*) в позицию итератора *pos*.
- Возвращает позицию первого вставленного элемента или итератор *pos*, если *beg*==*end* (до появления стандарта C++11 функция ничего не возвращала).
- Эта функция-член является шаблонной (см. раздел 3.2). Следовательно, элементы диапазона-источника могут иметь любой тип, который может быть преобразован в тип элементов контейнера.
- При работе с векторами эта операция делает итераторы и ссылки на элементы некорректными, если эти элементы предшествуют позиции *pos* и не выполняется перераспределение памяти.
- При работе с деками эта операция делает итераторы некорректными, а если позиция *pos* не совпадает с одним из концов, но некорректными становятся и все ссылки.
- Для списков функция работает по принципу "все или ничего".
- До появления стандарта C++11 использовался тип `iterator`, а не `const_iterator`, а возвращаемое значение имело тип `void`.
- Предусмотрена в векторе, деке, списке и строке.

8.7.3. Удаление элементов

`size_type` *контейнер*`::`**erase** `(const T&` *value*`)`

- Удаляет все элементы, эквивалентные аргументу *value*, из ассоциативного или упорядоченного контейнера.
- Возвращает количество удаленных элементов.
- Вызывает деструкторы удаляемых элементов.
- `T` — тип упорядоченных значений:
  - для (неупорядоченных) множеств и мультимножеств — это тип элементов;
  - для (неупорядоченных) отображений и мультиотображений — это тип ключей.

- Функция оставляет корректными итераторы и ссылки на другие элементы.
- Функция может генерировать исключение, если критерий сравнения или хеш-функция генерирует исключения.
- Предусмотрена в множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении.
- Для (последовательных) списков функция `remove()` делает то же самое (см. раздел 8.8.1). Для других контейнеров можно применять алгоритм `remove()` (см. раздел 11.7.1).

`iterator` *контейнер*`::`**erase** `(const_iterator` *pos*`)`

- Удаляет элемент, стоящий в позиции итератора *pos*.
- Возвращает позицию следующего элемента (или `end()`).
- Вызывает деструктор удаляемого элемента.
- Вызывающая сторона должна гарантировать корректность итератора *pos*. Например:

```
coll.erase(coll.end()); // ОШИБКА Þ непредсказуемые последствия
```

- Для векторов и деков операция может делать некорректными итераторы и ссылки на другие элементы. Для всех остальных контейнеров итераторы и ссылки на другие элементы остаются корректными.
- При работе со списками функция не генерирует исключений. При работе с векторами и деками функция не генерирует исключений при условии, что копирующий/перемещающий конструктор и оператор присваивания не генерируют исключений. При работе с ассоциативными и неупорядоченными контейнерами функция может генерировать исключение, если критерий сравнения или хеш-функция генерирует исключения.
- До появления стандарта C++11 для ассоциативных контейнеров типом возвращаемого значения был `void`, а вместо `const_iterator` использовался тип `iterator`.
- После появления стандарта C++11 вызов функции `erase()` для множеств, элементами которых являются итераторы, может оказаться неоднозначным. По этой причине в настоящее время стандарт C++11 предусматривает перегрузку для обоих вариантов: `erase(iterator)` и `erase(const_iterator)`.
- Предусмотрена в векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`iterator` *контейнер*`::`**erase** `(const_iterator` *beg*`, const_iterator` *end*`)`

- Удаляет э1деструкторы удаляемых элементов.
- Вызывающая сторона должна гарантировать, что позиции *beg* и *end* определяют корректный диапазон, являющийся частью контейнера.
- Для векторов и деков операция может делать некорректными итераторы и ссылки на другие элементы. Для всех остальных контейнеров итераторы и ссылки на другие элементы остаются корректными.

- При работе со списками функция не генерирует исключений. При работе с векторами и деками функция не генерирует исключений при условии, что копирующий/перемещающий конструктор и оператор присваивания не генерируют исключений. Для ассоциативных и неупорядоченных контейнеров функция может генерировать исключение, если критерий сравнения и функция хеширования генерируют исключения.
- До появления стандарта C++11 для ассоциативных контейнеров типом возвращаемого значения был `void`, а вместо `const_iterator` использовался тип `iterator`.
- Предусмотрен в векторе, деке, списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`void` *контейнер*`::`**pop_front** `()`

- Удаляет первый элемент контейнера.
- Эквивалентна выражению *контейнер*`.erase(`*контейнер*`.begin())` или (для последовательных списков) выражению *контейнер*`.erase_after(`*контейнер*`.before_begin())`
- Если контейнер пуст, последствия непредсказуемы. Следовательно, вызывающая сторона должна гарантировать, что контейнер содержит хотя бы один элемент (`!empty()`).
- Функция не генерирует исключений.
- Итераторы и ссылки на другие элементы остаются корректными.
- Предусмотрена в деке, списке и последовательном списке.

`void` *контейнер*`::`**pop_back** `()`

- Удаляет последний элемент контейнера.
- Эквивалентна выражению

контейнер`.erase(prev(`*контейнер*`.end()))`

- Если контейнер пуст, последствия непредсказуемы. Следовательно, вызывающая сторона должна гарантировать, что контейнер содержит хотя бы один элемент (`!empty()`).
- Функция не генерирует исключений.
- Итераторы и ссылки на другие элементы остаются корректными.
- Для строк предусмотрена начиная со стандарта C++11.
- Предусмотрена в векторе, деке, списке и строке.

`void` *контейнер*`::`**clear** `()`

- Удаляет все элементы (опустошает контейнер).
- Вызывает деструкторы удаляемых элементов.
- Делает некорректными все итераторы и ссылки на элементы контейнера.
- Для векторов, деков и строк функция делает некорректным запредельный итератор, который был возвращен функцией `end()` или `cend()`.
- Функция не генерирует исключений (до появления стандарта C++11 для векторов и деков функция могла генерировать исключения, если копирующий конструктор или операция присваивания генерировал исключения).

- Предусмотрен в векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

8.7.4. Изменение размера

`void` *контейнер*`::`**resize** `(size_type` *num*`)`
`void` *контейнер*`::`**resize** `(size_type` *num*`, const T&` *value*`)`

- Изменяют количество элементов до *num*.
- Если `size()` на входе равен *num*, функции не применяются.
- Если *num* на входе больше `size()`, создаются дополнительные элементы, которые добавляются в конец контейнера. Первая версия создает новые элементы, вызывая их конструкторы по умолчанию; вторая версия создает новые элементы как копии аргумента *value*.
- Если *num* на входе меньше `size()`, элементы удаляются с конца контейнера, чтобы размер контейнера стал равным *num*. В этом случае функции вызывают деструкторы удаляемых элементов.
- Для векторов и деков эта операция может делать некорректными итераторы и ссылки на другие элементы. Для всех остальных контейнеров итераторы и ссылки на другие элементы остаются корректными.
- Для векторов и деков эти функции работают по принципу "все или ничего", при условии, что конструктор или операция присваивания не генерирует исключений. Для списков и последовательных списков функции работают по принципу "все или ничего".
- До появления стандарта C++11 аргумент *value* передавался по значению.
- Для строк аргумент *value* передается по значению.
- Предусмотрены в векторе, деке, списке, последовательном списке и строке.

8.8. Специальные функции-члены для списков и последовательных списков

8.8.1. Специальные функции-члены для списков (и последовательных списков)

`void` *список*`::`**remove** `(const T&` *value*`)`
`void` *список*`::`**remove_if** `(UnaryPredicate` *op*`)`

- Функция `remove()` удаляет все элементы, имеющие значение, равное *value*.
- Функция `remove_if()` удаляет все элементы, для которых унарный предикат *op*`(`*elem*`)` возвращает `true`.
- Предикат *op* не должен изменять состояние во время вызова функции. Подробности приведены в разделе 10.1.4.
- Обе функции вызывают деструкторы удаляемых элементов.

- Порядок оставшихся элементов остается прежним.
- Функции представляют собой специальные версии алгоритма `remove()`, рассматриваемого в разделе 11.7.1.
- `T` — тип элементов контейнера.
- Дальнейшие подробности изложены в разделе 7.5.2.
- Функции могут генерировать исключения, только если сравнение элементов может генерировать исключения.
- Предусмотрены в списке и последовательном списке.

`void` *список*`::`**unique** `()`
`void` *список*`::`**unique** `(BinaryPredicate` *op*`)`

- Удаляет последующие дубликаты элементов (последовательного) списка, так что значение каждого элемента отличается от значения следующего элемента.
- Первый вариант удаляет все элементы, значения которых равны значениям предыдущих элементов.
- Второй вариант удаляет все элементы, следующие за элементом *e*, для которых бинарный предикат *op*(*elem*, *e*) возвращает `true`. Иначе говоря, этот предикат используется для сравнения элемента не с его предшественником, а с предыдущим элементом, который не был удален.
- Предикат *op* не должен изменять состояние во время вызова функции. Подробности приведены в разделе 10.1.4.
- Обе функции вызывают деструкторы удаляемых элементов.
- Функции представляют собой специальные версии алгоритмов `unique()` (см. раздел 11.7.2).
- Функции могут генерировать исключения, только если критерий сравнения элементов может генерировать исключения.
- Предусмотрены в списке и последовательном списке.

`void` *список*`::`**splice** `(const_iterator` *pos*`,` *список*`&` *source*`)`
`void` *список*`::`**splice** `(const_iterator` *pos*`,` *список*`&&` *source*`)`

- Перемещают все элементы из списка *source* в список `*this` и вставляют их перед позицией итератора *pos*.
- После выполнения вызова список *source* остается пустым.
- Если списки *source* и `*this` идентичны, последствия непредсказуемы. Следовательно, вызывающая сторона должна гарантировать, что *source* — другой список. Для перемещения элементов в одном и том же списке следует использовать функции `splice()`, показанные ниже.
- Вызывающая сторона должна гарантировать, что итератор *pos* имеет корректную позицию в списке `*this`; в противном случае последствия непредсказуемы.
- Указатели, итераторы и ссылка на члены списка *source* остаются корректными. Следовательно, после выполнения функций они ссылаются на список `this`.
- Функции не генерируют исключений.

- Вторая версия доступна начиная со стандарта C++11. До появления стандарта C++11 вместо `const_iterator` использовался тип `iterator`.
- Предусмотрены в списке.

`void` *список*`::`**splice** `(const_iterator` *pos*`,` *список*`&` *source*`,`
`const_iterator` *sourcePos*`)`
`void` *список*`::`**splice** `(const_iterator` *pos*`,` *список*`&&` *source*`,`
`const_iterator` *sourcePos*`)`

- Перемещает элемент из позиции *sourcePos* списка *source* в список `*this` и вставляет его перед позицией итератора *pos*.
- Списки *source* и `*this` могут быть идентичными. В этом случае элемент перемещается в списке.
- Если список *source* не совпадает со списком `*this`, после выполнения операции он будет содержать на один элемент меньше.
- Вызывающая сторона должна гарантировать, что итератор *pos* установлен на корректную позицию в списке `*this`, итератор *sourcePos* является корректным итератором в списке *source*, а итератор *sourcePos* не совпадает с итератором *source*.`end()`; в противном случае последствия непредсказуемы.
- Указатели, итераторы и ссылка на члены списка *source* остаются корректными. Следовательно, после выполнения функций они ссылаются на стыковочный элемент в списке `this`.
- Функции не генерируют исключений.
- Вторая версия доступна начиная со стандарта C++11. До появления стандарта C++11 вместо `const_iterator` использовался тип `iterator`.
- Предусмотрены в списке.

`void` *список*`::`**splice** `(const_iterator` *pos*`,` *список*`&` *source*`,`
`const_iterator` *sourceBeg*`,` `const_iterator` *sourceEnd*`)`
`void` *список*`::`**splice** `(const_iterator` *pos*`,` *список*`&&` *source*`,`
`const_iterator` *sourceBeg*`,` `const_iterator` *sourceEnd*`)`

- Перемещают элементы диапазона [*sourceBeg*,*sourceEnd*) списка *source* в список `*this` и вставляют их перед позицией итератора *pos*.
- Списки *source* и `*this` могут быть идентичными. В этом случае итератор *pos* не должен быть частью перемещаемого диапазона, а элементы перемещаются внутри списка.
- Если список *source* не совпадает со списком `*this`, после выполнения операции он будет содержать на несколько элементов меньше.
- Вызывающая сторона должна гарантировать, что итератор *pos* занимает корректную позицию в списке `*this`, а итераторы *sourceBeg* и *sourceEnd* определяют корректный диапазон, являющийся частью списка *source*; в противном случае последствия непредсказуемы.
- Указатели, итераторы и ссылки на другие элементы списка *source* остаются корректными. Следовательно, после выполнения функций они ссылаются на стыковочный элемент в списке `this`.

- Функции не генерируют исключений.
- Вторая версия доступна, начиная со стандарта C++11. До появления стандарта C++11 вместо `const_iterator` использовался тип `iterator`.
- Предусмотрены в списке.

`void` *список*`::`**sort** `()`
`void` *список*`::`**sort** `(CompFunc` *cmpPred*`)`

- Упорядочивает элементы.
- Первая версия упорядочивает все элементы с помощью операции <.
- Вторая версия упорядочивает все элементы, применяя предикат *cmpPred* для сравнения двух элементов:
- *op*(*elem1*, *elem2*)
- Порядок элементов, имеющих одинаковые значения, остается устойчивым, если только не генерируется исключение.
- Эти функции представляют собой специальные версии алгоритмов `sort()` и `stable_sort()` (см. раздел 11.9.1).
- Предусмотрены в списке и последовательном списке.

`void` *список*`::`**merge** `(`*список*`&` *source*`)`
`void` *список*`::`**merge** `(`*список*`&&` *source*`)`
`void` *список*`::`**merge** `(`*список*`&` *source*`, CompFunc` *cmpPred*`)`
`void` *список*`::`**merge** `(`*список*`&&` *source*`, CompFunc` *cmpPred*`)`

- Объединяют все элементы (последовательного) списка *source* в списке `*this`.
- После выполнения операции список *source* остается пустым.
- Первые две версии в качестве критерия сортировки используют операцию <.
- Последние две версии используют предикат *cmpPred* в качестве необязательного критерия сортировки и для сравнения двух элементов:

  cmpPred(*elem*, *sourceElem*)

- Порядок элементов, имеющих одинаковые значения, остается постоянным.
- Если списки `*this` и *source* упорядочены на входе согласно критерию сравнения < или *cmpPred*, то результат — (последовательный) список — также упорядочен и эквивалентные элементы списка `*this` предшествуют эквивалентным элементам списка *source*. Строго говоря, стандарт требует, чтобы оба (последовательных) списка на входе были упорядочены. Однако на практике возможно объединение неупорядоченных списков. Впрочем, прежде чем использовать, эту возможность сначала следует проверить.
- Эти функции представляют собой специальную версию алгоритма `merge()` (см. раздел 11.10.2).
- Если критерии сравнения элементов не генерируют исключений, то функции работают по принципу "все или ничего".
- Предусмотрены в списке и последовательном списке.

`void` *список*`::`**reverse** `()`

- Изменяет порядок следования элементов (последовательного) списка.
- Представляет собой специальную версию алгоритма `reverse()` (см. раздел 11.8.1).
- Функция не генерирует исключений.
- Предусмотрена в списке и последовательном списке.

8.8.2. Специальные функции-члены, предназначенные только для последовательных списков

`iterator` *последовательный_список*`::`**before_begin** `()`
`const_iterator` *последовательный_список*`::`**before_begin** `() const`
`const_iterator` *последовательный_список*`::`**cbefore_begin** `() const`

- Возвращает итератор, установленный на позицию, предшествующую позиции первого элемента.
- Поскольку последовательный список не допускает обратного обхода, эта функция-член позволяет создать позицию для вставки нового первого элемента или удаления существующего первого элемента.
- Предусмотрена в последовательном списке.

`iterator` *последовательный_список*`::`**insert_after** `(const_iterator` *pos*`,`
`const T&` *value*`)`
`iterator` *последовательный_список*`::`**insert_after** `(const_iterator` *pos*`, T&&` *value*`)`

- Вставляет аргумент *value* непосредственно после позиции итератора *pos*.
- Первая версия копирует аргумент *value*.
- Вторая версия перемещает аргумент *value* в контейнер, так что после выполнения операции состояние аргумента *value* становится неопределенным.
- Возвращает позицию нового элемента.
- Функция работает по принципу "все или ничего".
- Передача итератора `end()` или `cend()` контейнера в качестве аргумента *pos* приводит к непредсказуемым последствиям.
- Предусмотрена в последовательном списке.

`iterator` *последовательный_список*`::`**emplace_after** `(const_iterator` *pos*`,` *args*`)`

- Вставляет новый элемент, инициализированный списком аргументов *args*, непосредственно после позиции итератора *pos*.
- Возвращает позицию нового элемента.
- Функция работает по принципу "все или ничего".[2]
- Передача итератора `end()` или `cend()` контейнера в качестве аргумента *pos* приводит к непредсказуемым последствиям.
- Предусмотрена в последовательном списке.

[2] В настоящее время стандарт ничего не говорит об этом, что, вероятно, является его недочетом.

`iterator` *последовательный_список*`::`**insert_after** `(const_iterator` *pos*`, size_type` *num*`, const T&` *value*`)`

- Вставляет *num* копий аргумента *value* непосредственно после позиции итератора *pos*.
- Возвращает позицию последнего вставленного элемента или итератор *pos*, если *num*==0.
- Функция работает по принципу "все или ничего".
- Передача итераторов `end()` или `cend()` контейнера в качестве аргумента *pos* приводит к непредсказуемым последствиям.
- Предусмотрена в последовательном списке.

`iterator` *последовательный_список*`::`**insert_after** `(const_iterator` *pos*`,` *список-инициализации*`)`

- Вставляет копии элементов *списка инициализации* непосредственно после позиции итератора *pos*.
- Возвращает позицию последнего вставленного элемента или итератор *pos*, если *список инициализации* пуст.
- Функция работает по принципу "все или ничего".
- Передача итераторов `end()` или `cend()` контейнера в качестве аргумента *pos* приводит к непредсказуемым последствиям.
- Предусмотрена в последовательном списке.

`iterator` *последовательный_список*`::`**insert_after** `(const_iterator` *pos*`, InputIterator` *beg*`, InputIterator` *end*`)`

- Вставляет копии всех элементов диапазона [*beg,end*) непосредственно после итератора *pos*.
- Возвращает позицию последнего вставленного элемента или итератор *pos*, если *beg==end*.
- Эта функция-член является шаблонной (см. раздел 3.2). Следовательно, элементы диапазона-источника могут иметь любой тип, который может быть преобразован в тип элементов контейнера.
- Функция работает по принципу "все или ничего".
- Передача итераторов `end()` или `cend()` контейнера в качестве аргумента *pos* приводит к непредсказуемым последствиям.
- Предусмотрена в последовательном списке.

`iterator` *последовательный_список*`::`**erase_after** `(const_iterator` *pos*`)`

- Удаляет элемент, стоящий непосредственно после позиции итератора *pos*.
- Возвращает позицию следующего элемента (или `end()`).
- Вызывает деструктор удаляемого элемента.
- Итераторы и ссылки на другие элементы остаются корректными.
- Вызывающая сторона должна гарантировать корректность итератора *pos*, исключая обращение к позиции `end()` и позиции, предшествующей `end()`.
- Функция не генерирует исключений.

- Передача итераторов `end()` или `cend()` контейнера в качестве аргумента *pos* приводит к непредсказуемым последствиям.
- Предусмотрена в последовательном списке.

`void` *последовательный_список*`::`**erase_after** `(const_iterator` *beg*`, const_iterator` *end*`)`

- Удаляет элементы диапазона (*beg,end*). Обратите внимание, что это *не* полуоткрытый диапазон, потому что из него исключены позиции *beg* и *end*. Например:

```
coll.erase(coll.before_begin(),coll.end()); // OK: стирает все элементы
```

- Возвращает позицию *end*.
- Вызывает деструкторы удаляемых элементов.
- Вызывающая сторона должна гарантировать, что позиции *beg* и *end* определяют корректный диапазон, являющийся частью контейнера.
- Функция не генерирует исключений.
- Итераторы и ссылки на другие элементы остаются корректными.
- Предусмотрена в последовательном списке.

`void` *последовательный_список*`::`**splice_after** `(const_iterator` *pos*`,` *последовательный_список*`&` *source*`)`
`void` *последовательный_список*`::`**splice_after** `(const_iterator` *pos*`,` *последовательный_список*`&&` *source*`)`

- Перемещают все элементы из списка *source* в список `*this` и вставляют их в позицию, непосредственно после итератора *pos*.
- После выполнения вызова список *source* становится пустым.
- Если списки *source* и `*this` идентичны, последствия непредсказуемы. Следовательно, вызывающая сторона должна гарантировать, что *source* — другой список. Для перемещения элементов в одном и том же списке следует использовать одну из версий функции `splice_after()`.
- Вызывающая сторона должна гарантировать, что итератор *pos* имеет корректную позицию в списке `*this`; в противном случае последствия непредсказуемы.
- Указатели, итераторы и ссылка на члены списка *source* остаются корректными. Следовательно, после выполнения функций они ссылаются на стыковочный элемент в списке `this`.
- Функции не генерируют исключений.
- Передача итератора `end()` или `cend()` контейнера в качестве аргумента *pos* приводит к непредсказуемым последствиям.
- Предусмотрены в последовательном списке.

`void` *последовательный_список*`::`**splice_after** `(const_iterator` *pos*`,` *последовательный_список*`&` *source*`, const_iterator` *sourcePos*`)`
`void` *последовательный_список*`::`**splice_after** `(const_iterator` *pos*`,` *последовательный_список*`&&` *source*`, const_iterator` *sourcePos*`)`

- Перемещает элемент из позиции *sourcePos* списка *source* в список `*this` и вставляет его непосредственно после итератора *pos*.

- Списки *source* и `*this` могут быть идентичными. В этом случае элемент перемещается внутри списка.
- Если список *source* не совпадает со списком `*this`, после выполнения операции он будет содержать на один элемент меньше.
- Вызывающая сторона должна гарантировать, что итератор *pos* установлен на корректную позицию в списке `*this`, итератор *sourcePos* является корректным итератором в списке *source*, а итератор *sourcePos* не совпадает с итератором *source*. `end()`; в противном случае последствия непредсказуемы.
- Указатели, итераторы и ссылка на члены списка *source* остаются корректными. Следовательно, после выполнения функций они принадлежат списку `this`.
- Передача итератора `end()` или `cend()` контейнера в качестве аргумента *pos* приводит к непредсказуемым последствиям.
- Предусмотрены в последовательном списке по стандарту C++11.

`void` *последовательный_список*`::`**splice_after** `(const_iterator` *pos*`,`
последовательный_список`&` *source*`, const_iterator` *sourceBeg*`,`
`const_iterator` *sourceEnd*`)`
`void` *последовательный_список*`::`**splice_after** `(const_iterator` *pos*`,`
последовательный_список`&&` *source*`, const_iterator` *sourceBeg*`,`
`const_iterator` *sourceEnd*`)`

- Перемещают элементы диапазона (*sourceBeg*,*sourceEnd*) списка *source* в список `*this` и вставляют их в позицию итератора *pos*. Обратите внимание, что это *не* полуоткрытый диапазон, потому что из него исключены позиции *SourceBeg* и *SourceEnd*. Например, следующий вызов перемещает все элементы списка `coll2` в начало списка `coll`.

```
coll.splice_after(coll.before_begin(), coll2,
                  coll2.before_begin(), coll2.end());
```

- Списки *source* и `*this` могут быть идентичными. В этом случае итератор *pos* не должен быть частью перемещаемого диапазона, а элементы перемещаются внутри списка.
- Если список *source* не совпадает со списком `*this`, после выполнения операции он будет содержать на несколько элементов меньше.
- Вызывающая сторона должна гарантировать, что итератор *pos* занимает корректную позицию в списке `*this`, а итераторы *sourceBeg* и *sourceEnd* определяют корректный диапазон, являющийся частью списка *source*; в противном случае последствия непредсказуемы.
- Указатели, итераторы и ссылки на другие элементы списка *source* остаются корректными. Следовательно, после выполнения операции они принадлежат списку `this`.
- Функции не генерируют исключений.
- Передача итераторов `end()` или `cend()` контейнера в качестве аргумента *pos* приводит к непредсказуемым последствиям.
- Предусмотрены в последовательном списке.

8.9. Интерфейсы стратегий

8.9.1. Немодифицирующие вспомогательные функции

`size_type` *контейнер*`::`**capacity** `() const`

- Возвращает количество элементов контейнера, которые он может содержать без перераспределения памяти.
- Предусмотрена в векторе и строке.

`value_compare` *контейнер*`::`**value_comp** `() const`

- Возвращает объект, используемый в качестве критерия сравнения значений в ассоциативных массивах.
- В множествах и мультимножествах эквивалентна `key_comp()`.
- В отображениях и мультиотображениях представляет собой вспомогательный класс для критерия сравнения, сравнивающего только ключи из пар "ключ–значение".
- Предусмотрена в множестве, мультимножестве, отображении и мультиотображении.

`key_compare` *контейнер*`::`**key_comp** `() const`

- Возвращает критерий сравнения для ассоциативных контейнеров.
- Предусмотрена в множестве, мультимножестве, отображении и мультиотображении.

`key_equal` *контейнер*`::`**key_eq** `() const`

- Возвращает критерий сравнения для неупорядоченных контейнеров.
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

`hasher` *контейнер*`::`**hash_function** `() const`

- Возвращает хеш-функцию для неупорядоченных контейнеров.
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

`float` *контейнер*`::`**load_factor** `() const`

- Возвращает текущее среднее количество элементов в сегменте неупорядоченного контейнера.
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

`float` *контейнер*`::`**max_load_factor** `() const`

- Возвращает коэффициент максимального заполнения неупорядоченного контейнера. Контейнер автоматически выполняет повторное хеширование (увеличивает количество сегментов при необходимости), чтобы коэффициент нагрузки был меньше или равен этому числу.
- По умолчанию это число равно `1.0` и обычно должно уточняться (см. раздел 7.9.2).
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

8.9.2. Модифицирующие вспомогательные функции

`void` *контейнер*`::`**reserve** `(size_type` *num*`)`

- Резервирует внутреннюю память как минимум для *num* элементов.
- Для векторов эта функция может только увеличивать емкость. Следовательно, если *num* меньше или равно текущей емкости, функция никаких действий не выполняет. Для уменьшения емкости векторов используется функция `shrink_to_fit()` (см. пример в разделе 7.3.1).
- Для неупорядоченных контейнеров
  - этот вызов эквивалентен вызову `rehash(ceil(`*num*`/`*max_load_factor*`))` (функция `ceil()` возвращает число, округленное в сторону увеличения);
  - эта операция делает некорректными итераторы, изменяет порядок следования элементов, а также изменяет сегменты, в которых расположены элементы; указатели и ссылки на элементы остаются корректными.
- Для строк аргумент *num* является необязательным (по умолчанию: `0`), и вызов представляет собой несвязанный запрос на уменьшение размера, если число *num* меньше реальной емкости.
- Эта функция может делать некорректными итераторы, а также (для векторов и строк) ссылки и указатели на элементы. Однако гарантируется, что повторное выделение памяти после вызова функции `reserve()` не произойдет, пока после очередной вставки размер не превысит число *num*. Таким образом, функция `reserve()` может повышать быстродействие и способствовать сохранению корректности ссылок, указателей и итераторов (см. раздел 7.3.1).
- Генерирует исключение `length_error` (см. раздел 4.3.1), если *num*`>max_size()`, или соответствующее исключение, если выделение памяти закончилось неудачей.
- Доступна для неупорядоченных контейнеров начиная со стандарта C++11.
- Предусмотрена в векторе, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`void` *контейнер*`::`**shrink_to_fit** `()`

- Уменьшает объем внутренней памяти до объема, требуемого для хранения в точности имеющегося количества элементов.
- Этот вызов представляет собой несвязывающий запрос, т.е. реализации могут игнорировать его, чтобы обеспечить специализированную оптимизацию. Таким образом, нет гарантии, что после выполнения этой операции выражение `capacity() == size()` вернет значение `true`.
- Эта операция может делать некорректными ссылки, указатели и итераторы, установленные на элементы.
- Доступна начиная со стандарта C++11. Пример уменьшения емкости векторов до появления стандарта C++11 см. в разделе 7.3.1.
- Предусмотрена в векторе, деке и строке.

`void` *контейнер*`::`**rehash** `(size_type` *bnum*`)`

- Изменяет количество сегментов неупорядоченного контейнера как минимум до *bnum*.
- Эта операция делает некорректными итераторы, изменяет порядок элементов, и изменяет сегменты, в которых размещены элементы. Указатели и ссылки на элементы остаются корректными.
- Если генерируется исключение, которое отличается от исключения, генерируемого контейнерной хеш-функцией или функцией сравнения, операция не выполняет никаких действий.
- Для неупорядоченных мультимножеств и мультиотображений повторное хеширование сохраняет относительный порядок следования эквивалентных элементов.
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`void` *контейнер*`::`**max_load_factor** `(float` *loadFactor*`)`

- Устанавливает коэффициент максимального заполнения неупорядоченного контейнера равным *loadFactor*.
- Аргумент *loadFactor* интерпретируется как подсказка, поэтому реализации могут уточнять это значение по своему усмотрению.
- Эта операция может вызвать повторное хеширование, делающее некорректными итераторы, изменяющее порядок элементов и сегменты, в которых размещены элементы. Указатели и ссылки на элементы остаются корректными.
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

8.9.3. Сегментный интерфейс для неупорядоченных контейнеров

`size_type` *контейнер*`::`**bucket_count** `() const`

- Возвращает текущее количество сегментов неупорядоченного контейнера.
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`size_type` *контейнер*`::`**max_bucket_count** `() const`

- Возвращает максимально возможное количество сегментов неупорядоченного контейнера.
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`size_type` *контейнер*`::`**bucket** `(const key_type` *key*`) const`

- Возвращает индекс сегмента, в котором можно найти элементы с ключом, эквивалентным аргументу *key*, если таковые существуют.
- Возвращаемое значение лежит в диапазоне [0,`bucket_count()`).

- Если функция `bucket_count()` возвращает нуль, возвращаемое значение функции `bucket()` является неопределенным.
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`size_type` *контейнер*`::`**bucket_size** `(size_type` *bucketIdx*`) const`

- Возвращает количество элементов в сегменте с индексом *bucketIdx*.
- Если индекс *bucketIdx* не является корректным значением из диапазона [0,`bucket_count()`), последствия непредсказуемы.
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

`local_iterator` *контейнер*`::`**begin** `(size_type` *bucketIdx*`)`
`const_local_iterator` *контейнер*`::`**begin** `(size_type` *bucketIdx*`) const`
`const_local_iterator` *контейнер*`::`**cbegin** `(size_type` *bucketIdx*`) const`

- Все три версии возвращают итератор, установленный в начало всех элементов (позицию первого элемента) сегмента с индексом *bucketIdx*.
- Если сегмент пустой, то этот вызов эквивалентен вызовам *контейнер*`::end(`*bucketIdx*`)` или *контейнер*`::cend(`*bucketIdx*`)` соответственно.
- Если индекс *bucketIdx* не является корректным значением из диапазона [0,`bucket_count()`), последствия непредсказуемы.
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении.

`local_iterator` *контейнер*`::`**end** `(size_type` *bucketIdx*`)`
`const_local_iterator` *контейнер*`::`**end** `(size_type` *bucketIdx*`) const`
`const_local_iterator` *контейнер*`::`**cend** `(size_type` *bucketIdx*`) const`

- Все три версии возвращают итератор, установленный в конец всех элементов (позицию после последнего элемента) сегмента с индексом *bucketIdx*.
- Если сегмент пустой, то этот вызов эквивалентен вызовам *контейнер*`::begin(`*bucketIdx*`)` или *контейнер*`::cbegin(`*bucketIdx*`)` соответственно.
- Если индекс *bucketIdx* не является корректным значением из диапазона [0,`bucket_count()`), последствия непредсказуемы.
- Предусмотрены в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении.

8.10. Функции для выделения памяти

Все контейнеры STL можно использовать совместно со специальной моделью памяти, определенной распределителем памяти (см. главу 19). В этом разделе описываются члены, поддерживающие работу распределителя.

8.10.1. Основные члены распределителя памяти

контейнер::**allocator_type**

- Тип распределителя.
- Предусмотрен в векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке

`allocator_type` *контейнер*::**get_allocator** `() const`

- Возвращает модель памяти контейнера.
- Предусмотрен в векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

8.10.2. Конструкторы для необязательных параметров распределителя памяти

`explicit` *контейнер*::*контейнер* `(const Allocator&` *alloc*`)`

- Создает новый пустой контейнер, использующий модель памяти *alloc*.
- Предусмотрен в векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер::*контейнер* `(const CompFunc&` *cmpPred*`, const Allocator&` *alloc*`)`

- Создает новый пустой контейнер с предикатом *cmpPred*, используемым как критерий сортировки, и моделью памяти *alloc*.
- Критерий сортировки должен определять *строгое слабое упорядочение* (см. раздел 7.7).
- Предусмотрен в множестве, мультимножестве, отображении и мультиотображении.

контейнер::*контейнер* `(size_type` *bnum*`, const Hasher&` *hasher*`,`
`const KeyEqual&` *eqPred*`, const Allocator&` *alloc*`)`

- Создает новый пустой контейнер, имеющий не менее *bnum* сегментов, функцию хеширования *hasher*, предикат *eqPred*, используемый как критерий для идентификации одинаковых значений, и модель памяти *alloc*.
- Предусмотрен в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

контейнер::*контейнер* `(`*список-инициализации*`, const Allocator&` *alloc*`)`

- Создает новый контейнер, использующий модель памяти *alloc* и инициализированный элементами *списка инициализации*.

- Доступен начиная со стандарта C++11.
- Предусмотрен в векторе, деке, списке, последовательном списке и строке.

контейнер::*контейнер* (*список-инициализации*, `const CompFunc&` *cmpPred*, `const Allocator&` *alloc*)

- Создает новый пустой контейнер с предикатом *cmpPred*, используемым как критерий сортировки, и моделью памяти *alloc*, инициализированной элементами *списка инициализации*.
- Критерий сортировки должен определять *строгое слабое упорядочение* (см. раздел 7.7).
- Доступен начиная со стандарта C++11.
- Предусмотрен в множестве, мультимножестве, отображении и мультиотображении.

контейнер::*контейнер* (*список-инициализации*, `size_type` *bnum*, `const Hasher&` *hasher*, `const KeyEqual&` *eqPred*, `const Allocator&` *alloc*)

- Создает новый пустой контейнер, имеющий не менее *bnum* сегментов, функцию хеширования *hasher*, предикат *eqPred*, используемый как критерий для идентификации одинаковых значений, модель памяти *alloc* и инициализированный элементами *списка инициализации*.
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

контейнер::*контейнер* (`const` *контейнер*`&` *c*, `const Allocator&` *alloc*)
контейнер::*контейнер* (*контейнер*`&&` *c*, `const Allocator&` *alloc*)

- Создает новый контейнер, использующий модель памяти *alloc* и инициализированный скопированными/перемещенными элементами существующего контейнера *c*.
- Вызывает копирующий/перемещающий конструктор каждого элемента контейнера *c*.
- После вызова второй версии контейнер *c* остается корректным, но имеет неопределенное значение.
- Доступен начиная со стандарта C++11.
- Предусмотрен в векторе, деке, списке, последовательном списке, множестве, мультимножестве, отображении, мультиотображении, неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении, неупорядоченном мультиотображении, строке.

контейнер::*контейнер* (`size_type` *num*, `const T&` *value*, `const Allocator&` *alloc*)

- Создает контейнер, содержащий *num* элементов и использующий модель памяти *alloc*.
- Элементы создаются как копии аргумента *value*.
- `T` — тип элементов контейнера. Для строк аргумент *value* передается по значению.
- Предусмотрен в векторе, деке, списке, последовательном списке и строке.

контейнер::*контейнер* (`InputIterator` *beg*, `InputIterator` *end*, `const Allocator&` *alloc*)

- Создает контейнер, инициализированный всеми элементами диапазона [*beg*,*end*) и использующий модель памяти *alloc*.

- Эта функция-член является шаблонной (см. раздел 3.2). Следовательно, элементы диапазона-источника могут иметь любой тип, который может быть преобразован в тип элементов контейнера.
- Предусмотрен в векторе, деке, списке, последовательном списке и строке.

контейнер::*контейнер* (InputIterator *beg*, InputIterator *end*, const CompFunc& *cmpPred*, const Allocator& *alloc*)

- Создает контейнер, имеющий критерий сортировки *cmpPred* и инициализированный всеми элементами диапазона [*beg*,*end*) и использующий модель памяти *alloc*.
- Эта функция-член является шаблонной (см. раздел 3.2). Следовательно, элементы диапазона-источника могут иметь любой тип, который может быть преобразован в тип элементов контейнера.
- Критерий сортировки должен определять *строгое слабое упорядочение* (см. раздел 7.7).
- Предусмотрен в множестве, мультимножестве, отображении и мультиотображении.

контейнер::*контейнер* (InputIterator *beg*, InputIterator *end*, size_type *bnum*, const Hasher& *hasher*, const KeyEqual& *eqPred*, const Allocator& *alloc*)

- Создает контейнер, имеющий не менее *bnum* сегментов, функцию хеширования *hasher*, критерий эквивалентности *eqPred*, использующий модель памяти *alloc* и инициализированный всеми элементами диапазона [*beg*,*end*).
- Предусмотрена в неупорядоченном множестве, неупорядоченном мультимножестве, неупорядоченном отображении и неупорядоченном мультиотображении.

Глава 9

Итераторы STL

В главе подробно описываются итераторы, в частности, категории итераторов, операции над ними, адаптеры итераторов и пользовательские итераторы.

9.1. Заголовочные файлы для итераторов

Все контейнеры определяют свои собственные типы итераторов, поэтому для использования итераторов в контейнерах нет необходимости в специальном заголовочном файле. Однако некоторые определения специальных итераторов, например обратных итераторов, и некоторые функции для итераторов содержатся в заголовочном файле `<iterator>`. Некоторые реализации включают этот заголовочный файл при использовании любых контейнеров, чтобы определить для них типы обратных итераторов. Однако это препятствует переносимости программ. Следовательно, для того чтобы использовать необычные контейнерные итераторы и их типы, необходимо вставить в программу этот файл.

9.2. Категории итераторов

Итераторы — это объекты, которые могут обходить элементы последовательности с помощью универсального интерфейса, напоминающего интерфейс обычных указателей (см. раздел 6.3). Итераторы следуют принципу чистой абстракции: все, что *ведет* себя как итератор, *является* итератором. Однако итераторы наделены разными возможностями, которые имеют большое значение, потому что некоторые алгоритмы предъявляют к итераторам особые требования. Например, алгоритмы сортировки требуют, чтобы итераторы обеспечивали прямой доступ, потому что в противном случае быстродействие программы может снизиться. По этой причине итераторы разделяются на категории (рис. 9.1). Возможности этих категорий перечислены в табл. 9.1 и обсуждаются в следующих подразделах.

Считывающие итераторы, которые могут также записывать данные, называются *модифицирующими* (например, *модифицирующий последовательный итератор*).

9.2.1. Итераторы вывода

Итераторы вывода могут перемещаться только вперед и допускают запись. Следовательно, с их помощью можно лишь последовательно присваивать элементам новые значения. Итератор вывода нельзя использовать для повторного обхода одного и того же диапазона. Фактически даже не гарантируется, что возможно повторное присваивание без увеличения итератора. Эти итераторы предназначены для записи значения в "черную дыру" следующим образом:

```
while (...) {
   *pos = ...;  // присваиваем значение
   ++pos;       // перемещаемся вперед (готовимся к следующему присваиванию)
}
```

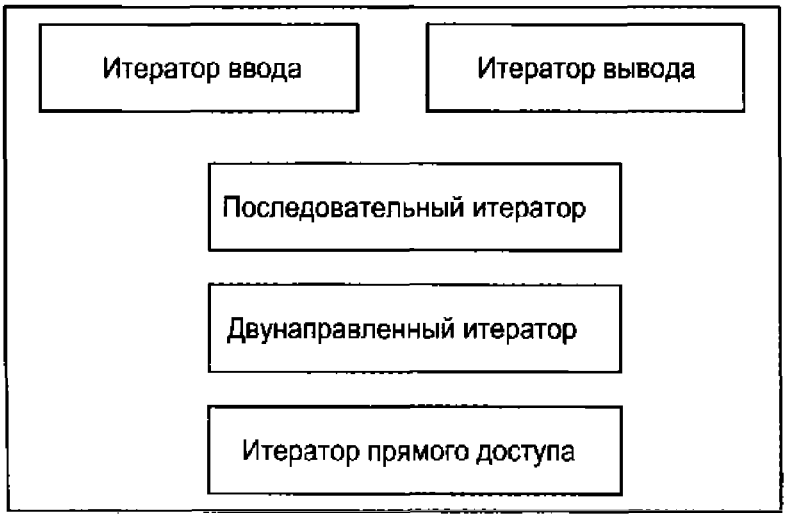


Рис. 9.1. Категории итераторов

Таблица 9.1. Обзор возможностей итераторов

Категория итераторов	Возможность	Поддержка
Итератор вывода	Запись в прямом направлении	Поток вывода, итератор вставки
Итератор ввода	Чтение в прямом направлении	Поток ввода
Однонаправленный итератор	Чтение в прямом направлении	Последовательный список, неупорядоченные контейнеры
Двунаправленный итератор	Чтение в прямом и обратном направлениях	Список, множество, мультимножество, отображение, мультиотображение
Итератор произвольного доступа	Чтение с произвольным доступом	Массив, вектор, дек, строка, массив в стиле языка C

В табл. 9.2 перечислены допустимые операции над итераторами вывода. Единственное допустимое использование операции * — в левой части оператора присваивания.

Таблица 9.2. Операции над итераторами вывода

Выражение	Результат
**iter = val*	Записывает значение *val* в то место, на которое ссылается итератор
++iter	Перемещает итератор вперед (возвращает новую позицию)
iter++	Перемещает итератор вперед (возвращает старую позицию)
TYPE (iter)	Копирует итератор (копирующий конструктор)

К итераторам вывода не применяются операции сравнения. Невозможно проверить, является ли итератор вывода корректным и была ли запись успешной. Можно только записывать, записывать и записывать значения. Как правило, конец записи определяется дополнительным требованием к конкретным итераторам вывода.

Часто итераторы могут и читать, и записывать значения. По этой причине все итераторы, выполняющие чтение, могут иметь дополнительную возможность записи. В этом случае они называются *модифицирующими*.

Типичным примером "чистого" итератора вывода является итератор, выполняющий запись в стандартный поток вывода (например, на экран или на принтер). Если вы используете два итератора вывода для записи на экран, то второе слово следует за первым, а не заменяет его. Другим типичным примером итераторов вывода являются итераторы вставки. Итераторы вставки — это итераторы, вставляющие значения в контейнеры. Присвоить им значение — значит вставить его. При записи второго значения первое не заменяется; второе значение просто вставляется, как и первое. Итераторы вставки рассматриваются в разделе 9.4.2.

Все константные итераторы, предусмотренные в контейнерах, а также их функции-члены `cbegin()` и `cend()` не являются итераторами вывода, потому что они не позволяют выполнять запись в позиции, на которые указывают эти итераторы.

9.2.2. Итераторы ввода

Итераторы ввода могут только последовательно обходить элемент за элементом, выполняя чтение. Следовательно, они возвращают значения поэлементно. Операции над итераторами ввода приведены в табл. 9.3.

Таблица 9.3. Операции над итераторами ввода

Выражение	Результат
**iter*	Обеспечивает доступ для чтения к текущему элементу
iter -> member	Обеспечивает доступ для чтения к члену текущего элемента
++iter	Перемещает итератор вперед (возвращает новую позицию)
iter++	Перемещает итератор вперед
iter1 == iter2	Возвращает результат проверки того, что два итератора являются равными
iter1 != iter2	Возвращает результат проверки того, что два итератора не являются равными
TYPE(iter)	Копирует итератор (копирующий конструктор)

Итераторы ввода могут читать элементы только один раз. Следовательно, если вы копируете итератор ввода и выполняете чтение с помощью оригинального итератора и его копии, то можете получить разные значения.

Все итераторы, ссылающиеся на значения, которые подлежат обработке, должны быть итераторами ввода. Однако обычно они могут больше. Типичным примером "чистого" итератора ввода является итератор, выполняющий чтение из стандартного потока ввода, которым, как правило, служит клавиатура. Одно и то же значение нельзя прочитать дважды. Как только слово считано из потока ввода, итератор переходит к следующему слову.

Для итераторов ввода операции == и != выполняют лишь проверку, равен ли итератор *запредельному итератору* (past-the-end iterator). Это необходимо потому, что операции над итераторами ввода обычно выполняются следующим образом:

```
InputIterator pos, end;

while (pos != end) {
```

```
    ... // доступ только для чтения с помощью *pos
    ++pos;
}
```

Нет гарантии, что два разных итератора, которые оба не равны запредельному итератору, при сравнении окажутся разными, если они ссылаются на разные позиции. (Это требование вводится начиная с однонаправленных итераторов.)

Отметим также, что итераторы ввода не требуют, чтобы оператор *iter*++ что-то возвращал; однако обычно он возвращает старую позицию итератора.

Всегда следует отдавать предпочтение префиксному оператору инкремента, а не постфиксному, потому что в общем случае он работает быстрее. Это объясняется тем, что префиксному инкременту не приходится возвращать старое значение, которое должно храниться во временном объекте. Следовательно, для любого итератора `pos` (и любого абстрактного типа) следует выбрать

```
++pos // хорошо и быстро
```

а не

```
pos++ // хорошо, но не так быстро
```

То же самое касается операторов декремента, если таковые определены (для итераторов ввода они не предусмотрены).

9.2.3. Однонаправленные итераторы

Однонаправленные итераторы — это итераторы ввода, обеспечивающие дополнительные гарантии при чтении в прямом направлении. Операции над однонаправленными итераторами приведены в табл. 9.4.

Таблица 9.4. Операции над однонаправленными итераторами

Выражение	Результат
**iter*	Обеспечивает доступ к текущему элементу
iter -> *member*	Обеспечивает доступ к члену текущего элемента
++*iter*	Перемещает итератор вперед (возвращает новую позицию)
iter++	Перемещает итератор вперед (возвращает старую позицию)
iter1 == *iter2*	Возвращает результат проверки того, что два итератора являются равными
iter1 != *iter2*	Возвращает результат проверки того, что два итератора не являются равными
TYPE ()	Создает итератор (конструктор по умолчанию)
TYPE (*iter*)	Копирует итератор (копирующий конструктор)
iter1 = *iter2*	Присваивает итератор

В отличие от итератора ввода, гарантируется, что для двух однонаправленных итераторов, ссылающихся на один и тот же элемент, операция == возвращает значение `true` и что после инкремента обоих итераторов они будут ссылаться на один и тот же элемент. Рассмотрим пример:

```
ForwardIterator pos1, pos2;

pos1 = pos2 = begin;  // оба итератора ссылаются на один и тот же элемент
if (pos1 != end) {
   ++pos1;            // итератор pos1 смещается на один элемент вперед
   while (pos1 != end) {
      if (*pos1 == *pos2) {
          ...         // обработка соседних дубликатов
          ++pos1;
          ++pos2;
      }
   }
}
```

Однонаправленные итераторы предусмотрены в следующих объектах и типах:

- класс `forward_list<>`;
- неупорядоченные контейнеры.

В то же время в неупорядоченных контейнерах библиотеки разрешают использовать вместо однонаправленных двунаправленные итераторы (см. раздел 7.9.1).

Однонаправленный итератор, удовлетворяющий требованиям итератора вывода, является *модифицирующим* однонаправленным итератором и может использоваться как для чтения, так и для записи.

9.2.4. Двунаправленные итераторы

Двунаправленные итераторы — это однонаправленные итераторы, обладающие дополнительной возможностью перемещаться по элементам в обратном направлении. Следовательно, они предусматривают оператор декремента для перемещения назад (табл. 9.5).

Таблица 9.5. Дополнительные операции над двунаправленными итераторами

Выражение	Результат
--iter	Перемещает итератор назад (возвращает новую позицию)
iter--	Перемещает итератор назад (возвращает старую позицию)

Двунаправленные итераторы предусмотрены в следующих объектах и классах:

- класс `list<>`;
- ассоциативные контейнеры.

Двунаправленный итератор, удовлетворяющий требованиям итератора вывода, является *модифицирующим* двунаправленным итератором и может использоваться как для чтения, так и для записи.

9.2.5. Итераторы произвольного доступа

Итераторы произвольного доступа обладают всеми возможностями двунаправленных итераторов и произвольным доступом к элементам. Следовательно, они реализуют операции *арифметики итераторов* (по аналогии с *арифметикой обычных указателей*). Иначе

говоря, к ним можно прибавлять и вычитать смещения, вычислять их разности и сравнивать итераторы с помощью операторов сравнения, например < и >. В табл. 9.6 приведены дополнительные операции над итераторами прямого доступа.

Таблица 9.6. Операции над итераторами прямого доступа

Выражение	Результат
iter[*n*]	Обеспечивает доступ к элементу с индексом *n*
iter+=*n*	Перемещает итератор на *n* элементов вперед (или назад, если *n* отрицательное)
iter-=*n*	Перемещает итератор на *n* элементов назад (или вперед, если *n* отрицательное)
iter+*n*	Возвращает итератор, установленный на элемент, расположенный на *n* позиций вперед
n+*iter*	Возвращает итератор, установленный на элемент, расположенный на *n* позиций вперед
iter-*n*	Возвращает итератор, установленный на элемент, расположенный на *n* позиций назад
iter1-*iter2*	Возвращает расстояние между итераторами *iter1* и *iter2*
iter1<*iter2*	Возвращает результат проверки того, что итератор *iter1* предшествует итератору *iter2*
iter1>*iter2*	Возвращает результат проверки того, что итератор *iter2* предшествует итератору *iter1*
iter1<=*iter2*	Возвращает результат проверки того, что итератор *iter2* не предшествует итератору *iter1*
iter1>=*iter2*	Возвращает результат проверки того, что итератор *iter1* предшествует итератору *iter2*

Итераторы произвольного доступа предусмотрены в следующих объектах и типах:

- контейнеры с произвольным доступом (`array`, `vector`, `deque`);
- строки (`string`, `wstring`);
- обычные массивы в стиле C (указатели).

Следующая программа иллюстрирует специальные возможности итераторов произвольного доступа:

```
// iter/itercategory1.cpp

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

int main()
{
   vector<int> coll;

   // вставляем элементы от -3 до 9
   for (int i=-3; i<=9; ++i) {
        coll.push_back (i);
```

```
    }

    // выводим на экран количество элементов,
    // вычисляя расстояние между концом и началом вектора
    // - ПРИМЕЧАНИЕ: к итераторам применяется оператор -
    cout << "number/distance: " << coll.end()-coll.begin() << endl;

    // выводим на экран все элементы
    // - ПРИМЕЧАНИЕ: к итераторам применяется оператор <, а не !=
    vector<int>::iterator pos;
    for (pos=coll.begin(); pos<coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;

    // выводим на экран все элементы
    // - ПРИМЕЧАНИЕ: к итераторам применяется оператор [], а не *
    for (int i=0; i<coll.size(); ++i) {
        cout << coll.begin()[i] << ' ';
    }
    cout << endl;

  // выводим на экран каждый второй элемент
  // - ПРИМЕЧАНИЕ: используется оператор +=
  for (pos = coll.begin(); pos < coll.end()-1; pos += 2) {
      cout << *pos << ' ';
    }
    cout << endl;
}
```

Программа выводит следующий результат.

```
number/distance: 13
-3 -2 -1 0 1 2 3 4 5 6 7 8 9
-3 -2 -1 0 1 2 3 4 5 6 7 8 9
-3 -1 1 3 5 7
```

Эта программа не работает с (последовательными) списками, (неупорядоченными) множествами и (неупорядоченными) отображениями, потому что все операции, упомянутые в примечаниях, предусмотрены только для итераторов произвольного доступа. В частности, следует помнить, что оператор < можно использовать в качестве критерия остановки цикла только при работе с итераторами произвольного доступа.

Отметим, что следующее выражение в последнем цикле требует, чтобы вектор `coll` содержал хотя бы один элемент:

```
pos < coll.end()-1
```

Если коллекция пустая, выражение `coll.end()-1` может вернуть позицию, предшествующую `coll.begin()`. Сравнение может по-прежнему выполняться, но, строго говоря, перемещение итератора на позицию, предшествующую началу контейнера, приводит к непредсказуемым последствиям. Аналогично выражение `pos += 2` может привести к непредсказуемым последствиям, если оно перемещает итератор за позицию `end()`. Таким образом, изменение последнего цикла на следующий фрагмент может оказаться очень

опасным, поскольку приводит к непредсказуемым последствиям, если коллекция содержит нечетное количество элементов (рис. 9.2).

```
for (pos = coll.begin(); pos < coll.end(); pos += 2) {
   cout << *pos << ' ';
}
```

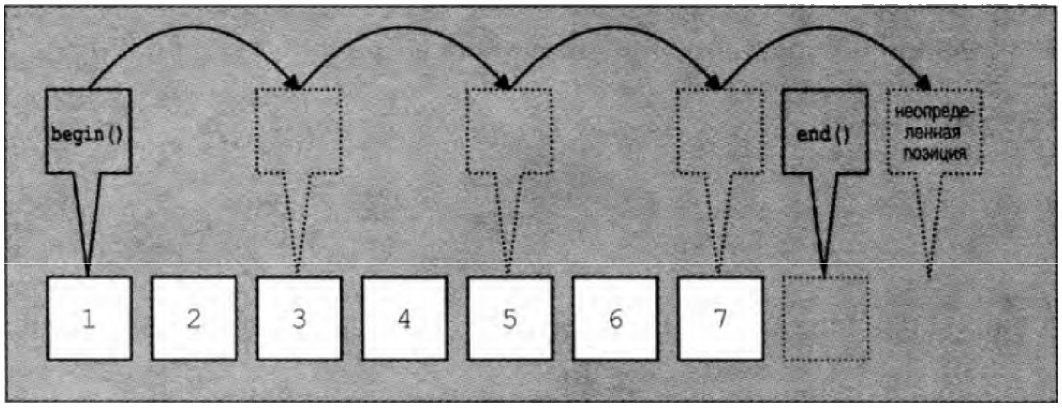


Рис. 9.2. Перемещение итераторов вправо на несколько позиций

Итератор произвольного доступа, удовлетворяющий требованиям итератора вывода, является *модифицирующим* итератором произвольного доступа и может использоваться как для чтения, так и для записи.

9.2.6. Проблема инкремента и декремента итераторов вектора

Использование операторов инкремента и декремента итераторов создает странную проблему. В общем случае можно применять инкремент и декремент к временным итераторам. Однако для объектов класса `array`, векторов и строк на некоторых платформах такие программы могут не компилироваться. Рассмотрим следующий пример:

```
std::vector<int> coll;

...
// сортируем, начиная со второго элемента
// - версия, не допускающая перенос на другие компьютеры
if (coll.size() > 1) {
    std::sort(++coll.begin(), coll.end());
}
```

В зависимости от реализации компиляция выражения `++coll.begin()` может закончиться неудачей. Однако если вы используете, скажем, дек, а не вектор, то компиляция пройдет успешно.

Причина этой странной проблемы заключается в том, что итераторы векторов, массивов и строк могут быть реализованы, как обычные указатели. Как и при работе с любыми другими элементарными типами, такими как указатели, модифицировать временные значения нельзя. В то же время для структур и классов это разрешено. Таким образом, если итератор реализован как обычный указатель, компиляция заканчивается неудачей; если же он реализован в виде класса, компиляция проходит успешно.

Вследствие этого предыдущий код всегда работает со всеми контейнерами (за исключением класса `array`, векторов и строк), потому что для контейнеров нельзя реализовать итераторы как обычные указатели. Однако для массивов, векторов и строк работоспособность кода зависит от реализации. Часто используются обычные указатели. Если же вы, к примеру, используете "безопасную версию" библиотеки STL (что случается все чаще), то итераторы реализуются как классы.

Для обеспечения переносимости стандарт C++11 предусмотрел вспомогательную функцию `next()` (см. раздел 9.3.2), поэтому можно написать следующий код:

```
std::vector<int> coll;

...
// сортируем начиная со второго элемента
// - переносимо начиная со стандарта C++11
if (coll.size() > 1) {
    std::sort(std::next(coll.begin()), coll.end());
}
```

До появления стандарта C++11 следовало использовать вспомогательный объект:

```
std::vector<int> coll;

...
// сортируем начиная со второго элемента
// - переносимо до появления стандарта C++11
if (coll.size() > 1) {
    std::vector<int>::iterator beg = coll.begin();
    std::sort(++beg, coll.end());
}
```

Эта проблема не такая страшная, как кажется. Непредвиденных последствий не наступает, потому что проблема выявляется на этапе компиляции. И все же досадно тратить время на ее решение.

9.3. Вспомогательные функции для работы с итераторами

Стандартная библиотека содержит несколько вспомогательных функций для работы с итераторами: `advance()`, `next()`, `prev()`, `distance()` и `iter_swap()`. Первые четыре функции наделяют все итераторы некоторыми возможностями, которыми обладают только итераторы произвольного доступа: перемещаться на один элемент вперед (или назад) и вычислять разность между итераторами. Последняя вспомогательная функция позволяет обменивать значения двух итераторов.

9.3.1. Функция advance()

Функция `advance()` переносит итератор вперед на указанное аргументом количество позиций. Следовательно, эта функция позволяет итератору перемещаться вперед (или назад) на несколько элементов.

```
#include <iterator>
void advance (InputIterator& pos, Dist n)
```

- Переносит итератор ввода *pos* на *n* элементов вперед (или назад).
- Для двунаправленных итераторов и итераторов произвольного доступа число *n* может быть отрицательным, задавая перемещение назад.
- `Dist` — шаблонный тип. Обычно он должен быть целочисленным типом, потому что к нему применяются такие операции, как `<`, `++`, `--`, и сравнения с `0`.
- Отметим, что функция `advance()` *не* проверяет, заходит ли она за позицию `end()` последовательности (она не может это проверить, потому что итераторы вообще не знают ничего о контейнерах, которые они обходят). Следовательно, вызов этих функций может привести к непредсказуемым последствиям, потому что результат применения операции `++` к концу последовательности не определен.

Благодаря свойствам итераторов (см. раздел 9.5) эта функция всегда использует наилучшую реализацию, зависящую от категории итератора. Для итераторов произвольного доступа она просто выполняет вызов *pos*+=*n*. Следовательно, для таких итераторов функция `advance()` имеет константную сложность. Для всех других итераторов она *n* раз выполняет оператор ++*pos* (или --*pos*, если *n* отрицательное). Таким образом, для всех других категорий итераторов функция `advance()` имеет линейную сложность.

Для того чтобы иметь возможность изменять контейнеры и типы итераторов, следует использовать функцию `advance()`, а не операцию +=. Однако, делая это, следует учитывать риск непреднамеренного снижения производительности программы. Дело в том, что вы не обнаружите никакого снижения производительности, пока не воспользуетесь контейнерами, не предоставляющими итераторы произвольного доступа (низкое быстродействие — одна из причин, по которым оператор += предусмотрен только для итераторов произвольного доступа). Отметим также, что функция `advance()` ничего не возвращает, а оператор += возвращает новую позицию и поэтому может быть частью более крупного выражения. Рассмотрим пример использования функции `advance()`:

```
// iter/advance1.cpp

#include <iterator>
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
 {
   list<int> coll;

   // вставляем элементы от 1 до 9
   for (int i=1; i<=9; ++i) {
        coll.push_back(i);
   }

   list<int>::iterator pos = coll.begin();

   // выводим на экран текущий элемент
```

```
    cout << *pos << endl;

    // перемещаемся на три элемента вперед
    advance (pos, 3);

    // выводим на экран текущий элемент
    cout << *pos << endl;

    // перемещаемся на один элемент назад
    advance (pos, -1);

    // выводим на экран текущий элемент
    cout << *pos << endl;
}
```

В этой программе функция `advance()` позволяет переместить итератор `pos` на три элемента вперед и на один элемент назад. Следовательно, результат работы программы будет таким:

```
1
4
3
```

Другой способ использования функции `advance()` — игнорирование некоторых данных, получаемых итераторами при чтении потока ввода. См. пример в разделе 9.4.3.

9.3.2. Функции `next()` и `prev()`

В соответствии со стандартом C++11 две эти вспомогательные функции позволяют перемещать итератор в следующую или предыдущую позицию:

```
#include <iterator>
ForwardIterator next (ForwardIterator pos)
ForwardIterator next (ForwardIterator pos, Dist n)
```

- Возвращают позицию прямого итератора *pos*, в которую он попал бы, если бы переместился на одну или *n* позиций вперед.
- Для двунаправленных итераторов и итераторов произвольного доступа число *n* может быть отрицательным и приводить к перемещению в предыдущие позиции.
- `Dist` — это тип `std::iterator_traits<ForwardIterator>::difference_type`.
- При выполнении функции выполняется вызов `advance(pos,n)` для внутреннего временного объекта.
- Функция `next()` *не* проверяет факт выхода за позицию `end()` последовательности. Следовательно, именно вызывающая сторона должна гарантировать корректность результата.

```
#include <iterator>
BidirectionalIterator prev (BidirectionalIterator pos)
BidirectionalIterator prev (BidirectionalIterator pos, Dist n)
```

- Возвращают позицию двунаправленного итератора *pos*, в которую он попал бы, если бы переместился на одну или *n* позиций назад.
- Для перехода на последующие позиции число *n* может быть отрицательным.
- `Dist` — это тип `std::iterator_traits<BidirectionalIterator>::difference_type`.
- При выполнении функции выполняется вызов `advance(`*pos*`, -`*n*`)` для внутреннего временного объекта.
- Отметим, что функция `prev()` *не* проверяет факт выхода за позицию `begin()` последовательности. Следовательно, именно вызывающая сторона должна гарантировать корректность результата.

Эти функции позволяют, например, обойти коллекцию, проверяя значения следующих элементов:

```
auto pos = coll.begin();
while (pos != coll.end() && std::next(pos) != coll.end()) {
       ...
       ++pos;
}
```

Это особенно полезно, учитывая, что прямые и двунаправленные итераторы не предусматривают операции + и -. В противном случае нам всегда требовался бы временный объект

```
auto pos = coll.begin();
auto nextPos = pos;
++nextPos;
while (pos != coll.end() && nextPos != coll.end()) {
       ...
       ++pos;
       ++nextPos;
}
```

или код, использующий исключительно итераторы произвольного доступа.

```
auto pos = coll.begin();
while (pos != coll.end() && pos+1 != coll.end()) {
       ...
       ++pos;
}
```

Не забудьте проверить, что позиция является корректной, перед тем как использовать ее (по этой причине мы сначала проверяем, не равен ли итератор `pos` итератору `coll.end()`, прежде чем перейти на следующую позицию).

Другое приложение функций `next()` и `prev()` — возможность не писать выражения, такие как `++coll.begin()`, для работы со вторым элементом коллекции. Проблема заключается в том, что использование выражения `++coll.begin()` вместо `std::next(coll.begin())` может вызвать ошибку компиляции (см. раздел 9.2.6).

Третье применение функции `next()` — это работа с классом `forward_list` и функцией `before_begin()` (см. раздел 7.6.2).

9.3.3. Функция distance()

Функция distance() предназначена для вычисления разности между двумя итераторами.

```
#include <iterator>
Dist distance (InputIterator pos1, InputIterator pos2)
```

- Возвращает расстояние между итераторами ввода *pos1* и *pos2*.
- Оба итератора должны ссылаться на элементы, принадлежащие одному и тому же контейнеру.
- Если итераторы не являются итераторами произвольного доступа, то позиция итератора *pos2* должна быть достижимой из позиции итератора *pos1*; иначе говоря, итератор *pos2* должен иметь ту же позицию или позицию, расположенную правее.
- Тип возвращаемого значения *Dist* — это тип разности, соответствующий типу итератора.

  ```
  iterator_traits<InputIterator>::difference_type
  ```

- Подробности изложены в разделе 9.5.

Используя дескрипторы итератора, эта функция выбирает наилучшую реализацию в соответствии с категорией итератора. Для итераторов произвольного доступа эта функция просто возвращает значение выражения *pos2-pos1*. Таким образом, для таких итераторов функция distance() имеет константную сложность. Для всех других категорий итераторов выполняется многократный инкремент итератора *pos1*, пока он не достигнет позиции итератора *pos2*, и после этого возвращается количество выполненных инкрементов. Следовательно, для всех остальных категорий итераторов функция distance() имеет линейную сложность. Итак, функция distance() имеет низкую производительность для всех итераторов, кроме итераторов произвольного доступа. Подумайте, как избежать ее применения в своей программе.

Реализация функции distance() описана в разделе 9.5.1. Ее использование демонстрирует следующий пример:

```
// iter/distance1.cpp

#include <iterator>
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
   list<int> coll;

   // вставляем элементы от -3 до 9
   for (int i=-3; i<=9; ++i) {
        coll.push_back(i);
   }
```

```
   // ищем элемент, имеющий значение 5
   list<int>::iterator pos;
   pos = find (coll.begin(), coll.end(), // диапазон
              5);                        // значение
   if (pos != coll.end()) {
   // вычисляем и выводим разность между текущей позицией и началом
      cout << "difference between beginning and 5: "
           << distance(coll.begin(),pos) << endl;
   }
   else {
      cout << "5 not found" << endl;
   }
}
```

После того как итератору `pos` будет присвоена возвращенная функцией `find()` позиция элемента, имеющего значение 5, функция `distance()` использует эту позицию для вычисления разности между нею и началом контейнера. Результат работы программы выглядит так:

```
difference between beginning and 5: 8
```

Для того чтобы иметь возможность изменять типы итераторов и контейнеров, следует использовать функцию `distance()`, а не `-`. Однако если вы используете функцию `distance()`, то не обнаружите снижения производительности, пока не перейдете от итераторов произвольного доступа к другим видам итераторов.

При вычислении разности между итераторами, которые не являются итераторами произвольного доступа, будьте внимательны. Первый итератор должен ссылаться на элемент, расположенный до, а не после элемента, на который ссылается второй итератор. В противном случае последствия будут непредсказуемыми. Если вам неизвестно, какой из итераторов предшествует другому, следует вычислить расстояние между ними и началом контейнера, а затем вычислить разность между этими расстояниями. Однако для этого необходимо знать, к какому контейнеру относится итератор. В противном случае возможны непредсказуемые последствия. Дополнительные аспекты этой проблемы, связанные с подынтервалами, изложены в разделе 6.4.1.

9.3.4. Функция `iter_swap()`

Это простая вспомогательная функция, предназначенная для обмена значений, на которые ссылаются два итератора.

```
#include <algorithm>
void iter_swap (ForwardIterator1 pos1, ForwardIterator2 pos2)
```

- Обменивает значения, на которые ссылаются итераторы *pos1* и *pos2*.
- Итераторы не обязаны иметь одинаковый тип. Однако значения должны допускать присваивание.

Рассмотрим простой пример (функция `PRINT_ELEMENTS()` введена в разделе 6.6):

```
// iter/iterswap1.cpp

#include <iostream>
```

```
#include <list>
#include <algorithm>
#include <iterator>
#include "print.hpp"
using namespace std;

int main()
{
   list<int> coll;

   // вставляем элементы от 1 до 9
   for (int i=1; i<=9; ++i) {
        coll.push_back(i);
   }

   PRINT_ELEMENTS(coll);

   // Обмениваем первое и второе значения
   iter_swap (coll.begin(), next(coll.begin()));
   PRINT_ELEMENTS(coll);

   // Обмениваем первое и последнее значения
   iter_swap (coll.begin(), prev(coll.end()));
   PRINT_ELEMENTS(coll);
}
```

Результат работы программы выглядит следующим образом:

```
1 2 3 4 5 6 7 8 9
2 1 3 4 5 6 7 8 9
9 1 3 4 5 6 7 8 2
```

Отметим, что функции `next()` и `prev()` предусмотрены стандартом C++11, а использование вместо них операторов `++` и `--` допускается не для каждого контейнера.

```
vector<int> coll;
...
iter_swap (coll.begin(), ++coll.begin()); // ОШИБКА: может не компилироваться
...
iter_swap (coll.begin(), --coll.end());   // ОШИБКА: может не компилироваться
```

Эта проблема подробно описана в разделе 9.2.6.

9.4. Адаптеры итераторов

В разделе описываются адаптеры итераторов, содержащиеся в стандартной библиотеке C++. Эти специальные итераторы позволяют алгоритмам выполнять обратный обход в режиме вставки и работать с потоками.

9.4.1. Обратные итераторы

Обратные итераторы (reverse iterators) переопределяют операции инкремента и декремента так, что их семантика становится обратной. Следовательно, если использовать эти итераторы вместо обычных, алгоритмы будут выполняться в обратном направлении.

Большинство контейнерных классов — все, за исключением последовательных списков и неупорядоченных контейнеров, — а также строки обеспечивают возможность использования обратных итераторов для обхода своих элементов. Рассмотрим следующий пример:

```
// iter/reviter1.cpp

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

void print (int elem)
{
   cout << elem << ' ';
}

int main()
{
   // создаем список элементов от 1 до 9
   list<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

   // выводим все элементы на экран в обычном порядке
   for_each (coll.begin(), coll.end(), // диапазон
             print);                   // операция
   cout << endl;

   // выводим все элементы на экран в обратном порядке
   for_each (coll.rbegin(), coll.rend(), // диапазон
             print);                     // операция
   cout << endl;
}
```

Функции-члены `rbegin()` и `rend()` возвращают обратный итератор. Аналогично функциям `begin()` и `end()`, эти итераторы определены на полуоткрытом диапазоне. Однако они действуют в обратном направлении.

- Функция-член **`rbegin()`** возвращает позицию первого элемента при обратном обходе. Следовательно, она возвращает позицию последнего элемента.
- Функция-член **`rend()`** возвращает позицию, расположенную за последним элементом при обратном обходе. Следовательно, она возвращает позицию, *предшествующую* первому элементу.

Результат работы программы выглядит следующим образом:

```
1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1
```

Стандарт C++11 предусматривает соответствующие функции-члены `crbegin()` и `crend()`, возвращающие обратные итераторы только для чтения. Поскольку они позволяют только читать элементы, их можно (и нужно) использовать в этом примере:

```
// выводим все элементы в обратном порядке
for_each (coll.crbegin(), coll.crend(), // диапазон
          print);                       // операция
```

Итераторы и обратные итераторы

Обычные итераторы можно превратить в обратные. Естественно, эти итераторы должны быть двунаправленными, но обратите внимание, что логическая позиция итератора при его преобразовании перемещается. Рассмотрим следующую программу:

```
// iter/reviter2.cpp

#include <iterator>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
   // создаем список с элементами от 1 до 9
   vector<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

   // ищем позицию элемента, имеющего значение 5
   vector<int>::const_iterator pos;
   pos = find (coll.cbegin(), coll.cend(),
               5);

   // выводим на экран значение, на которое ссылается итератор pos
   cout << "pos: " << *pos << endl;

   // преобразовываем итератор pos в обратный итератор rpos
   vector<int>::const_reverse_iterator rpos(pos);

   // выводим на экран значение, на которое ссылается итератор rpos
   cout << "rpos: " << *rpos << endl;
}
```

Эта программа выводит на экран следующий результат:

```
pos: 5
rpos: 4
```

Таким образом, при выводе значения итератора и его преобразовании в обратный итератор это значение изменилось. Это не ошибка, а особенность! Это поведение является следствием того факта, что диапазоны являются полуоткрытыми. Для того чтобы определить все элементы в контейнере, необходимо использовать позицию, расположенную за последним элементом. Однако для обратного итератора этой позицией является позиция,

предшествующая первому элементу. К сожалению, эта позиция может не существовать. Контейнеры не обязаны гарантировать, что позиция, предшествующая первому элементу, является корректной. Представьте себе, что обычные строки и массивы тоже контейнеры, а язык программирования не гарантирует, что адресация массивов не начинается с нулевого адреса...

В результате разработчики обратных итераторов использовали трюк: они "физически" изменили на противоположный "принцип полуоткрытости". Физически в диапазоне определены обратные итераторы, начало *не включено* в диапазон, а конец *включен*. Однако логически итераторы работают как обычно. Следовательно, существует различие между физической позицией, определяющей элемент, на который ссылается итератор, и логической позицией, определяющей значение, на которое ссылается итератор (рис. 9.3). Вопрос заключается в том, что происходит во время превращения итератора в обратный итератор? Какую позицию сохраняет итератор: логическую (значение) или физическую (элемент)? Как показывает предыдущий пример, имеет место второй вариант. Таким образом, значение перемещается на предыдущий элемент (рис. 9.4).

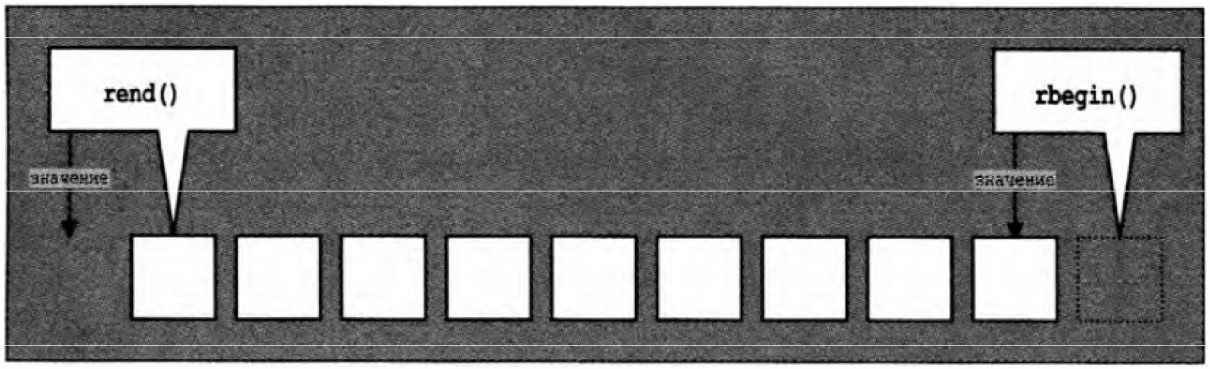


Рис. 9.3. Позиция и значение обратных итераторов

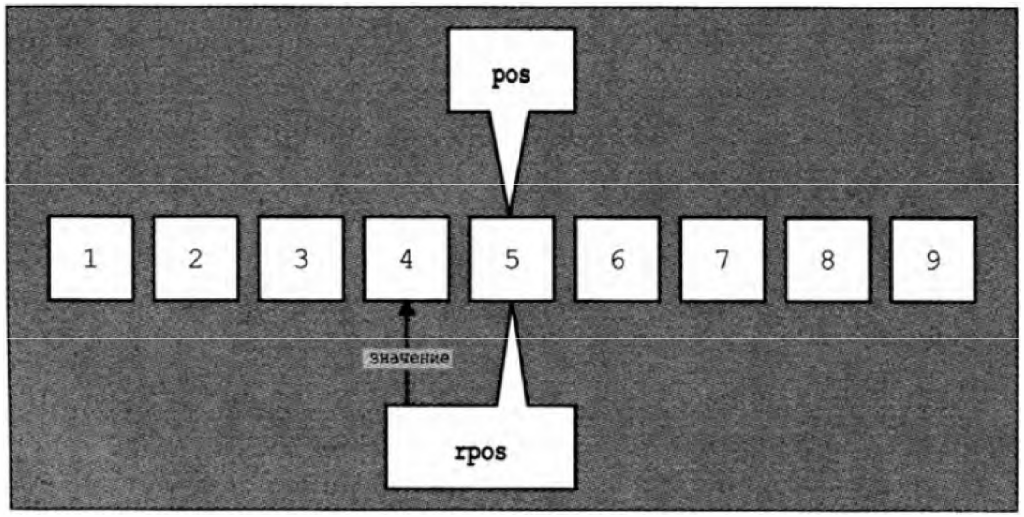


Рис. 9.4. Преобразование итератора pos в обратный итератор rpos

Вы поняли это решение? У него есть одно преимущество: не надо ничего делать, чтобы преобразовать диапазон, который определен двумя, а не одним итератором. Все элементы остаются корректными. Рассмотрим следующий пример:

```
// iter/reviter3.cpp

#include <iterator>
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;

void print (int elem)
{
   cout << elem << ' ';
}

int main()
{
   // создаем дек с элементами от  1 до 9
   deque<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

   // ищем позицию элемента со значением 2
   deque<int>::const_iterator pos1;
   pos1 = find (coll.cbegin(), coll.cend(), // диапазон
               2);                          // значение

   // ищем элемент со значением 7
   deque<int>::const_iterator pos2;
   pos2 = find (coll.cbegin(), coll.cend(), // диапазон
               7);                          // значение

   // выводим на экран все элементы из диапазона [pos1,pos2)
   for_each (pos1, pos2, // диапазон
             print);     // операция
   cout << endl;

   // превращаем итераторы в обратные
   deque<int>::const_reverse_iterator rpos1(pos1);
   deque<int>::const_reverse_iterator rpos2(pos2);

   // выводим на экран все элементы из диапазона [pos1,pos2) в обратном порядке
   for_each (rpos2, rpos1, // диапазон
             print);       // операция
   cout << endl;
}
```

Итераторы `pos1` и `pos2` определяют полуоткрытый диапазон, включая элемент со значением 2, но исключая элемент со значением 7. Когда итератор, описывающий этот диапазон, превращается в обратный, диапазон остается корректным и может обрабатываться в обратном порядке. Таким образом, программа выводит на экран следующие результаты:

```
2 3 4 5 6
6 5 4 3 2
```

Итак, функция-член `rbegin()` — это просто

контейнер`::reverse_iterator(end())`

а функция-член `rend()` — это просто

контейнер`::reverse_iterator(begin())`

Разумеется, константные итераторы могут преобразовываться в тип `const_reverse_iterator`.

Обратное преобразование обратных итераторов с помощью функции-члена base()

Обратные итераторы можно превратить обратно в обычные. Для этого обратные итераторы снабжены функцией-членом `base()`.

```
namespace std {
   template <typename Iterator>
   class reverse_iterator ... {
      ...
      Iterator base() const;
      ...
   };
}
```

Рассмотрим пример использования функции `base()`:

```
// iter/reviter4.cpp

#include <iterator>
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
   // создаем список с элементами от 1 до 9
   list<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

   // ищем позицию элемента, имеющего значение 5
   list<int>::const_iterator pos;
   pos = find (coll.cbegin(), coll.cend(), // диапазон
               5);                         // значение

   // выводим на экран этот элемент
   cout << "pos: " << *pos << endl;

   // превращаем итератор в обратный итератор
   list<int>::const_reverse_iterator rpos(pos);

   // выводим на экран значение элемента, на который ссылается обратный итератор
   cout << "rpos: " << *rpos << endl;

   // превращаем обратный итератор в обычный
   list<int>::const_iterator rrpos;
   rrpos = rpos.base();

   // выводим на экран значение элемента, на который ссылается обычный итератор
```

```
   cout << "rrpos: " << *rrpos << endl;
}
```

Эта программа выводит на экран следующие строки:

```
pos: 5
rpos: 4
rrpos: 5
```

Таким образом, преобразование итератора с помощью функции `base()`

```
*rpos.base()
```

эквивалентно его превращению в обратный итератор. Иначе говоря, физическая позиция (элемент итератора) сохраняется, а логическая (значение элемента) перемещается. Другой пример использования функции `base()` приведен в разделе 11.5.3.

9.4.2. Итераторы вставки

Итераторы вставки — это адаптеры итераторов, преобразовывающие присвоение нового значения во вставку этого нового значения. Используя итераторы вставки, алгоритмы могут вставлять элементы, а не заменять их. Все итераторы вставки относятся к категории итераторов вывода. Таким образом, они обеспечивают только возможность присваивать новые значения (см. раздел 9.2.1).

Функциональные возможности итераторов вставки

Как правило, алгоритмы присваивают значения итераторам получателя. Например, рассмотрим алгоритм `copy()` (он описан в разделе 11.6.1):

```
namespace std {
   template <typename InputIterator, typename OutputIterator>
   OutputIterator copy (InputIterator from_pos,  // начало источника
                        InputIterator from_end,  // конец источника
                        OutputIterator to_pos)   // начало получателя
   {
      while (from_pos != from_end) {
         *to_pos = *from_pos;   // копируем значения
         ++from_pos;            // инкрементируем итераторы
         ++to_pos;
      }
      return to_pos;
   }
}
```

Этот цикл выполняется, пока текущая позиция итератора источника не достигнет его конца. В цикле итератор источника `from_pos` присваивается итератору получателя `to_pos`, и оба итератора инкрементируются.

Представляет интерес часть, выполняющая присвоение нового значения:

```
   *to_pos = value
```

Итератор вставки преобразовывает это присваивание во вставку. Однако эта операция состоит из двух: во-первых, оператор * возвращает текущий элемент итератора, во-вторых, оператор = присваивает новое значение. Реализации итераторов вставки обычно используют двухэтапный трюк.

1. Оператор * реализуется как фиктивная операция, просто возвращающая `*this`. Таким образом, для итераторов вставки значение `*pos` эквивалентно `pos`.
2. Оператор присваивания реализуется так, что он превращается во вставку. Фактически итератор вставки вызывает функции-члены контейнера `push_back()`, `push_front()` или `insert()`.

Таким образом, чтобы вставить новый элемент с помощью итератора вставки, можно писать `pos = value`, а не `*pos = value`. Однако это касается деталей реализации итераторов ввода. Правильным выражением для присваивания нового значения является `*pos = value`.

Аналогично операция инкремента реализуется как фиктивная операция, просто возвращающая `*this`. Следовательно, модифицировать позицию итератора вставки невозможно. Все операции над итераторами вставки приведены в табл. 9.7.

Таблица 9.7. Операции над итераторами вставки

Выражение	Результат
**iter*	Фиктивный (возвращает *iter*)
iter = value	Вставляет значение
++iter	Фиктивный (возвращает *iter*)
iter++	Фиктивный (возвращает *iter*)

Виды итераторов вставки

Стандартная библиотека C++ содержит три вида итераторов вставки: итераторы вставки в конец, итераторы вставки в начало и общие итераторы вставки. Они различаются позициями, в которые выполняется вставка. Фактически каждый из них использует разные функции-члены контейнера. Следовательно, итератор вставки всегда должен быть инициализирован своим контейнером.

Каждый вид итератора вставки имеет удобную функцию для его создания и инициализации. В табл. 9.8 приведены разновидности итераторов вставки и их возможности.

Таблица 9.8. Виды итераторов вставки

Имя	Класс	Вызываемая функция	Создание
Итератор вставки в конец	`back_insert_iterator`	`push_back(value)`	`back_inserter(cont)`
Итератор вставки в начало	`front_insert_iterator`	`push_front(value)`	`front_inserter(cont)`
Общий итератор вставки	`insert_iterator`	`insert(pos,value)`	`inserter(cont,pos)`

Разумеется, контейнер должен иметь функцию-член, которую вызывает итератор вставки, в противном случае этот вид итераторов вставки использовать нельзя. По этой причине итераторы вставки в конец доступны только для векторов, деков, списков и строк, а итераторы вставки в начало — только для деков и списков.

Итераторы вставки в конец

Итератор вставки в конец (back inserter) добавляет значение в конец контейнера, вызывая функцию-член `push_back()` (описание этой функции см. в разделе 8.7.1). Функция `push_back()` доступна только для векторов, деков, списков и строк, поэтому это единственные контейнеры из стандартной библиотеки C++, для которых доступны итераторы вставки в конец.

Итератор вставки в конец должен быть инициализирован своим контейнером в момент своего создания. Функция `back_inserter()` обеспечивает удобный способ для этого. Использование итераторов вставки в конец демонстрирует следующий пример:

```
// iter/backinserter1.cpp

#include <vector>
#include <algorithm>
#include <iterator>
#include "print.hpp"
using namespace std;

int main()
{
   vector<int> coll;

   // создаем итератор вставки в конец для вектора coll
   // - неудобный способ
   back_insert_iterator<vector<int> > iter(coll);

   // вставка элементов с помощью интерфейса обычных итераторов
   *iter = 1;
   iter++;
   *iter = 2;
   iter++;
   *iter = 3;
   PRINT_ELEMENTS(coll);

   // создаем итератор вставки в конец и вставляем элемент
   // - удобный способ
   back_inserter(coll) = 44;
   back_inserter(coll) = 55;
   PRINT_ELEMENTS(coll);

   // снова используем итератор вставки в конец для добавления новых элементов
   // - резервируем достаточный объем памяти,
   // чтобы избежать ее повторного выделения
   coll.reserve(2*coll.size());
   copy (coll.begin(), coll.end(), // источник
         back_inserter(coll));     // получатель
```

```
    PRINT_ELEMENTS(coll);
}
```

Результат работы этой программы выглядит следующим образом:

```
1 2 3
1 2 3 44 55
1 2 3 44 55 1 2 3 44 55
```

Отметим, что перед вызовом функции `copy()` следует зарезервировать достаточный объем памяти. Причина заключается в том, что итератор вставки в конец вставляет элементы, что может сделать некорректными все остальные итераторы в том же векторе. Следовательно, если зарезервирован недостаточный объем памяти, алгоритм сделает некорректными переданные итераторы источника.

Строки также имеют контейнерный интерфейс STL, включая функцию-член `push_back()`. Следовательно, ее можно использовать для добавления символов в строку. Пример приведен в разделе 13.2.14.

Итераторы вставки в начало

Итератор вставки в начало (front inserter) вставляет значение в начало контейнера, вызывая функцию-член контейнера `push_front()` (подробное описание этой функции см. в разделе 8.7.1). Функция-член `push_front()` доступна только для деков, списков и последовательных списков, поэтому это единственные контейнеры из стандартной библиотеки C++, для которых доступны итераторы вставки в начало.

Итератор вставки в начало должен инициализироваться контейнером в момент своего создания. Функция `front_inserter()` обеспечивает удобный способ для этого. Использование итераторов вставки в начало демонстрирует следующий пример:

```
// iter/frontinserter1.cpp

#include <list>
#include <algorithm>
#include <iterator>
#include "print.hpp"
using namespace std;

int main()
{
    list<int> coll;
    // создаем итератор вставки в начало для списка coll
    // - неудобный способ
    front_insert_iterator<list<int> > iter(coll);

    // вставляем элементы с помощью интерфейса обычного итератора
    *iter = 1;
    iter++;
    *iter = 2;
    iter++;
    *iter = 3;
    PRINT_ELEMENTS(coll);
```

```
    // создаем итератор вставки в начало и вставляем элементы
    // - удобный способ
    front_inserter(coll) = 44;
    front_inserter(coll) = 55;
    PRINT_ELEMENTS(coll);

    // используем итератор для вставки всех элементов
    copy (coll.begin(), coll.end(), // источник
    front_inserter(coll));          // получатель
    PRINT_ELEMENTS(coll);
}
```

Результат работы программы выглядит следующим образом:

```
3 2 1
55 44 3 2 1
1 2 3 44 55 55 44 3 2 1
```

Отметим, что итератор вставки в начало вставляет несколько элементов в обратном порядке. Это объясняется тем, что следующий элемент всегда вставляется перед предыдущим.

Общие итераторы вставки

Общий итератор вставки (general inserter)[1] инициализируется двумя значениями: контейнером и позицией для вставки. Используя оба аргумента, итератор вставки вызывает функцию-член `insert()`, передавая ей заданную позицию как аргумент. Функция-член `inserter()` обеспечивает удобный способ для создания и инициализации общего итератора вставки.

Общий итератор вставки доступен для всех стандартных контейнеров (за исключением массивов и последовательных списков), поскольку все они имеют требуемую функцию-член `insert()` (см. раздел 8.7.1). Однако для ассоциативных и неупорядоченных контейнеров позиция используется только как подсказка, потому что в них конкретная позиция определяется значением элемента (см. раздел 8.7.1).

После вставки общий итератор вставки получает позицию нового вставленного элемента. В частности, выполняется такой код:

```
pos = container.insert(pos,value);
++pos;
```

Присваивание значения, возвращенного функцией `insert()`, гарантирует, что позиция итератора всегда является корректной. Без присваивания новой позиции в деках, векторах и строках общий итератор вставки делает сам себя некорректным. Это объясняется тем, что при каждой вставке все итераторы, ссылающиеся на контейнер, становятся некорректными (по крайней мере, это возможно).

Использование общих итераторов вставки иллюстрируется следующим примером:

```
// iter/inserter1.cpp

#include <set>
```

[1] Общий итератор вставки часто просто называют *итератор вставки*.

```
#include <list>
#include <algorithm>
#include <iterator>
#include "print.hpp"
using namespace std;

int main()
{
   set<int> coll;
   // создаем итератор вставки для множества coll
   // - неудобный способ
   insert_iterator<set<int> > iter(coll,coll.begin());

   // вставляем элементы с помощью интерфейса обычного итератора
   *iter = 1;
   iter++;
   *iter = 2;
   iter++;
   *iter = 3;
   PRINT_ELEMENTS(coll,"set: ");

   // создаем итератор вставки и вставляем элемент
   // - удобный способ
   inserter(coll,coll.end()) = 44;
   inserter(coll,coll.end()) = 55;
   PRINT_ELEMENTS(coll,"set: ");

   // используем итератор для вставки всех элементов в список
    list<int> coll2;
   copy (coll.begin(), coll.end(),       // источник
         inserter(coll2,coll2.begin())); // получатель
   PRINT_ELEMENTS(coll2,"list: ");

   // используем итератор вставки для повторной вставки всех элементов
   // в список перед вторым элементом
   copy (coll.begin(), coll.end(),         // источник
         inserter(coll2,++coll2.begin())); // получатель
   PRINT_ELEMENTS(coll2,"list: ");
}
```

Результат работы программы выглядит следующим образом:

```
set: 1 2 3
set: 1 2 3 44 55
list: 1 2 3 44 55
list: 1 1 2 3 44 55 2 3 44 55
```

Вызовы функции `copy()` показывают, что общий итератор вставки сохраняет порядок следования элементов. Второй вызов функции `copy()` использует конкретную позицию в диапазоне, который передается как аргумент.

Пользовательские итераторы вставки для ассоциативных контейнеров

Как указывалось ранее, при работе с ассоциативными контейнерами аргумент общих итераторов вставки, задающий позицию, рассматривается только как подсказка. Эта подсказка может повысить быстродействие, но может и снизить его. Например, если вставленные элементы следуют в обратном порядке, то эта подсказка может замедлить работу программы, так как поиск правильной позиции для вставки всегда будет начинаться с неправильной точки. Таким образом, неверная подсказка может оказаться хуже, чем отсутствие подсказки. Это хороший пример, демонстрирующий необходимость расширения стандартной библиотеки C++. Пример такого расширения описан в разделе 9.6.

9.4.3. Итераторы потоков

Итератор потока (stream iterator) — это адаптер итератора, позволяющий использовать поток в качестве источника или получателя данных для алгоритмов. В частности, итератор потока ввода можно использовать для чтения элементов из потока ввода, а итератор потока вывода — для записи элементов в поток вывода.

Особой формой итератора потока является итератор буфера потока (stream buffer iterator), который можно использовать для чтения или записи в буфер потока. Итераторы буферов потока обсуждаются в разделе 15.13.2.

Итераторы потока вывода

Итераторы потока вывода (ostream iterator) записывают присвоенные значения в поток вывода. С помощью итераторов потока вывода алгоритмы могут записывать данные непосредственно в потоки. Реализация итератора потока вывода использует ту же концепцию, что и реализация итераторов вставки (см. раздел 9.4.2). Единственное отличие заключается в том, что они преобразовывают присваивание нового значения в операцию вывода с помощью оператора <<. Таким образом, алгоритмы могут непосредственно записывать данные в потоки с помощью обычного интерфейса итераторов. Операции над итераторами потоков вывода приведены в табл. 9.9.

Таблица 9.9. Операции над итераторами `ostream`

Выражение	Результат
`ostream_iterator<`*T*`>(`*ostream*`)`	Создает итератор потока вывода для потока *ostream*
`ostream_iterator<`*T*`>(`*ostream*`,` *delim*`)`	Создает итератор потока вывода для потока *ostream* с разделителем, заданным строкой *delim* (строка *delim* имеет тип `const char*`)
**iter*	Фиктивная операция (возвращает итератор *iter*)
iter = *value*	Записывает значение в поток *ostream*: *ostream*<<*value* (после значения выводится разделитель *delim*, если таковой задан)
++*iter*	Фиктивная операция (возвращает итератор *iter*)
iter++	Фиктивная операция (возвращает итератор *iter*)

При создании итератора потока вывода необходимо указать поток вывода, в который будут записываться данные. С помощью необязательного аргумента можно указать строку, которая будет использоваться как разделитель между отдельными значениями. Без разделителя элементы будут просто непосредственно следовать друг за другом.

Итераторы потока вывода определяются для некоторого типа элементов `T`.

```
namespace std {
   template <typename T,
             typename charT = char,
             typename traits = char_traits<charT> >
class ostream_iterator;
}
```

Необязательный второй и третий шаблонные аргументы задают тип потока (см. раздел 15.2.1)[2].

Использование итераторов потоков вывода иллюстрируется следующим примером:

```
// iter/ostreamiter1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
   // создаем итератор потока вывода для потока cout
   // - значения разделяются символом перехода на новую строку
   ostream_iterator<int> intWriter(cout,"\n");

   // записываем элементы с помощью обычного интерфейса итераторов
   *intWriter = 42;
   intWriter++;
   *intWriter = 77;
   intWriter++;
   *intWriter = -5;

   // создаем коллекцию элементов от 1 до 9
   vector<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

   // записываем все элементы без разделителей
   copy (coll.cbegin(), coll.cend(),
         ostream_iterator<int>(cout));
   cout << endl;

   // записываем все элементы с разделителем " < "
   copy (coll.cbegin(), coll.cend(),
         ostream_iterator<int>(cout," < "));
   cout << endl;
}
```

[2] В более старых системах необязательные шаблонные аргументы для типа потока отсутствуют.

Результат работы этой программы выглядит следующим образом:

```
42
77
-5
123456789
1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 <
```

Отметим, что разделитель имеет тип `const char*`. Следовательно, если передать объект типа `string`, необходимо вызывать его функцию-член `c_str()` (см. раздел 13.3.6), чтобы преобразовать его в правильный тип. Например:

```
string delim;
...
ostream_iterator<int>(cout,delim.c_str());
```

Итераторы потоков ввода

Итераторы потоков ввода (istream iterator) — это противоположность итераторов потоков вывода. Итератор потока ввода читает элементы из потока ввода. Используя итераторы потока ввода, алгоритмы могут непосредственно читать данные из потоков. Однако итераторы потоков ввода немного сложнее итераторов потоков вывода (поскольку чтение, как обычно, немного сложнее, чем запись).

В момент создания итератор потока ввода инициализируется потоком ввода, из которого он будет читать данные. Затем с помощью обычного интерфейса итераторов ввода (см. раздел 9.2.2) итератор потока ввода читает элемент за элементом, используя оператор >>. Однако чтение может завершиться неудачей (из-за конца файла или ошибки), а диапазон источника данных для алгоритмов должен иметь "конечную позицию". Для решения этих проблем можно использовать *итератор конца потока* (end-of-stream iterator), созданный с помощью конструктора по умолчанию итератора потока ввода. Если чтение завершается неудачей, итератор потока ввода превращается в итератор конца потока. Следовательно, после каждой попытки чтения необходимо сравнивать итератор потока ввода с итератором конца потока, чтобы убедиться, что итератор имеет корректное значение. Все операции над итераторами потока ввода перечислены в табл. 9.10.

Таблица 9.10. Операции над итераторами `istream`

Выражение	Действие
`istream_iterator<`*T*`>()`	Создает итератор конца потока
`istream_iterator<`*T*`>(`*istream*`)`	Создает итератор потока ввода для потока *istream* (и может считывать первое значение)
**iter*	Возвращает считанное ранее значение (считывает первое значение, если оно не было считано конструктором)
iter->member	Возвращает член ранее прочитанного значения, если таковой имеется
++*iter*	Считывает следующее значение и возвращает его позицию
iter++	Считывает следующее значение и возвращает итератор для предыдущего значения
iter1 == *iter2*	Проверяет, равны ли итераторы *iter1* и *iter2*
iter1 != *iter2*	Проверяет, не равны ли итераторы *iter1* и *iter2*

Отметим, что конструктор итератора потока ввода открывает поток и обычно считывает первый элемент. В противном случае он не мог бы вернуть первый элемент при выполнении операции * после инициализации. Однако реализации могут отложить чтение первого значения до первого вызова операции *. Таким образом, итератор потока ввода не следует определять до того, как он потребуется.

Итераторы потоков ввода определены для некоторого элемента типа T.

```
namespace std {
   template <typename T,
             typename charT = char,
             typename traits = char_traits<charT>,
             typename Distance = ptrdiff_t>
class istream_iterator;
}
```

Необязательные второй и третий шаблонные аргументы задают тип используемого потока (см. раздел 15.2.1). Необязательный четвертый шаблонный элемент задает тип разности между итераторами[3].

Два потоковых итератора ввода считаются равными, если

- они являются итераторами конца потока и, следовательно, больше не могут читать данные, или
- оба итератора могут читать и использовать один и тот же поток.

Следующий пример демонстрирует операции, предусмотренные для потоковых итераторов ввода:

```
// iter/istreamiter1.cpp

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

int main()
{
   // создаем итератор потока ввода, читающий целые числа из потока cin
   istream_iterator<int> intReader(cin);

   // создаем итератор конца потока
   istream_iterator<int> intReaderEOF;
   // читаем лексемы с помощью потокового итератора ввода, пока это возможно
   // - записываем их дважды
   while (intReader != intReaderEOF) {
      cout << "once:       " << *intReader << endl;
      cout << "once again: " << *intReader << endl;
     ++intReader;
   }
}
```

Если выполнить эту программы с входными данными

```
1 2 3 f 4
```

[3] В более старых системах, не использовавших шаблонные параметры, заданные по умолчанию, необязательный четвертый шаблонный аргумент был вторым, а аргументы для типа потока отсутствовали.

то результат ее работы будет выглядеть так:

```
once: 1
once again: 1
once: 2
once again: 2
once: 3
once again: 3
```

Легко видеть, что ввод символа `f` завершает работу программы. Из-за ошибки формата поток теряет корректное состояние. Следовательно, итератор потока ввода `intReader` равен итератору конца потока `intReaderEOF`. Итак, условие цикла становится равным `false`.

Пример использования потоковых итераторов и функции advance()

Следующий пример демонстрирует использование обоих видов потоковых итераторов и вспомогательной функции `advance()`:

```
// iter/advance2.cpp

#include <iterator>
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
   istream_iterator<string> cinPos(cin);
   ostream_iterator<string> coutPos(cout," ");

   // пока при вводе не будет достигнут конец файла
   // - записываем каждую третью строку
   while (cinPos != istream_iterator<string>()) {

      // игнорируем две следующие строки
      advance (cinPos, 2);

      // читаем и записываем третью строку
      if (cinPos != istream_iterator<string>()) {
         *coutPos++ = *cinPos++;
      }
   }
   cout << endl;
}
```

Функция `advance()` перемещает итератор на другую позицию (см. раздел 9.3.1). Используемая с итераторами потоков ввода, функция `advance()` пропускает вводимые лексемы. Например, если поток ввода содержит следующие строки[4]:

[4] Благодарю Эндрю Кёнига (Andrew Koenig) за остроумный пример входных данных.

```
No one objects if you are doing
a good programming job for
someone whom you respect.
```

то на выводе получим:

```
objects are good for you
```

Не забудьте проверить, остался ли корректным итератор потока ввода после вызова функции `advance()` и перед обращением к значению с помощью выражения `*cinPos`. Применение операции `*` к итератору конца потока приводит к непредсказуемым последствиям.

Другие примеры, демонстрирующие использование алгоритмами итераторов потоков для чтения и записи данных, приведены в разделах 6.5.2, 11.6.1 и 11.7.2.

9.4.4. Итераторы перемещения

В стандарте C++11 имеется адаптер итератора, который превращает любое обращение к элементу в операцию перемещения. Например:

```
std::list<std::string> s;
...
std::vector<string> v1(s.begin(), s.end());          // копируем строки в v1

std::vector<string> v2(make_move_iterator(s.begin()), // перемещаем строки в v2
                       make_move_iterator(s.end()));
```

Эти итераторы позволяют алгоритмам перемещать, а не копировать элементы из одного диапазона в другой. Однако стоит заметить, что алгоритм `move()` делает то же самое (см. раздел 11.6.2).

В принципе использование *итератора перемещения* (move iterator) в алгоритмах целесообразно только в том случае, когда алгоритмы перемещают элементы из диапазона-источника в диапазон-получатель. Кроме того, необходимо гарантировать, что обращение к каждому элементу выполняется только один раз. В противном случае содержимое было бы перемещено несколько раз, что вызвало бы непредсказуемые последствия.

Отметим, что единственная категория итераторов, гарантирующая, что элементы считываются или обрабатываются только один раз, — это категория итераторов ввода (см. раздел 9.2.2). Таким образом, использование итераторов перемещения обычно целесообразно, только если алгоритм имеет источник, в котором требуется итератор ввода, а получатель использует итератор вывода. Единственное исключение — алгоритм `for_each()`, который можно использовать для обработки перемещенных элементов в заданном диапазоне (например, для их перемещения в новый контейнер).

9.5. Свойства итераторов

Разные категории итераторов (см. раздел 9.2) содержат итераторы с разными функциональными возможностями. Было бы полезно и даже необходимо переопределять поведение для разных категорий итераторов. Эту перегрузку можно выполнить с помощью дескрипторов и свойств итераторов (предусмотренных в заголовочном файле `<iterator>`).

Для каждой категории итераторов стандартная библиотека C++ предусматривает *дескриптор итератора* (iterator tag), который можно использовать как "метку" итератора.

```
namespace std {
   struct output_iterator_tag {
   };
   struct input_iterator_tag {
   };
   struct forward_iterator_tag
      : public input_iterator_tag {
   };
   struct bidirectional_iterator_tag
      : public forward_iterator_tag {
   };
   struct random_access_iterator_tag
      : public bidirectional_iterator_tag {
   };
}
```

Обратите внимание на то, что здесь используется наследование. Следовательно, любой однонаправленный итератор, например, *является разновидностью* итератора ввода. Однако дескриптор для однонаправленных итераторов выводится только из дескриптора итератора вывода. Таким образом, любой однонаправленный итератор *не является* разновидностью итератора вывода. Только *модифицирующие* однонаправленные итераторы одновременно удовлетворяют требованиям итераторов вывода (см. раздел 9.2), но для них нет отдельной категории.

При написании обобщенного кода может оказаться важной не только категория итератора. Например, может понадобиться тип элементов, на который ссылается итератор. Для этого стандартная библиотека C++ предоставляет специальную шаблонную структуру для определения *свойств итераторов* (iterator traits), которая содержит всю важную информацию об итераторе и используется как универсальный интерфейс для всех определений типов, необходимых для итераторов (категория, тип элементов и т.д.).

```
namespace std {
   template <typename T>
   struct iterator_traits {
      typedef typename T::iterator_category iterator_category;
      typedef typename T::value_type        value_type;
      typedef typename T::difference_type   difference_type;
      typedef typename T::pointer           pointer;
      typedef typename T::reference         reference;
   };
}
```

В этом шаблоне тип `T` обозначает тип итератора. Следовательно, можно написать код, который для любого итератора использует его категорию, тип элементов и т.д. Например. следующее выражение определяет тип значения для типа итератора `T`:

```
typename std::iterator_traits<T>::value_type
```

Эта структура имеет два преимущества:

- гарантирует, что итератор предоставляет определения всех типов;
- ее можно (частично) специализировать для (множества) специальных итераторов.

Второе преимущество реализовано для обычных указателей, которые также можно использовать как итераторы.

```
namespace std {
   template <typename T>
   struct iterator_traits<T*> {
      typedef T                          value_type;
      typedef ptrdiff_t                  difference_type;
      typedef random_access_iterator_tag iterator_category;
      typedef T*                         pointer;
      typedef T&                         reference;
   };
}
```

Таким образом, любой тип "указатель на элемент типа T" определен как итератор произвольного доступа. Кроме того, существует частичная специализация (const T*) для константных указателей.

Отметим, что итераторы вывода можно использовать только для записи. Следовательно, в итераторе вывода value_type, difference_type, pointer и reference можно определить как void.

9.5.1. Запись обобщенных функций для итераторов

Используя свойства итератора, можно писать обобщенные функции, которые выводят определения типов или используют разный код реализации в зависимости от категории итератора.

Использование типов итераторов

Простой пример использования свойств итератора — алгоритм, которому необходима временная переменная для элементов. Такое временное значение объявляется примерно так:

```
typename std::iterator_traits<T>::value_type tmp;
```

где T — тип итератора.

Другой пример — алгоритм циклического сдвига элементов.

```
template <typename ForwardIterator>
void shift_left (ForwardIterator beg, ForwardIterator end)
{
   // временная переменная для первого элемента
   typedef typename
      std::iterator_traits<ForwardIterator>::value_type value_type;

   if (beg != end) {
      // сохраняем значение первого элемента
      value_type tmp(*beg);

      // смещаем следующие значения
      ...
}
}
```

Использование категории итератора

Для использования разных реализаций для разных категорий итераторов необходимо сделать следующее.

- Разрешить шаблонной функции вызывать другую функцию, передавая категорию итератора как дополнительный аргумент. Например:

```
template <typename Iterator>
inline void foo (Iterator beg, Iterator end)
{
    foo (beg, end,
         std::iterator_traits<Iterator>::iterator_category());
}
```

- Реализовать эту другую функцию для произвольной категории итератора, имеющей специальную реализацию, которая не выводится из другой категории итераторов. Например:

```
// foo() для двунаправленных итераторов
template <typename BiIterator>
void foo (BiIterator beg, BiIterator end,
          std::bidirectional_iterator_tag)
{
    ...
}

// foo() для итераторов произвольного доступа
template <typename RaIterator>
void foo (RaIterator beg, RaIterator end,
          std::random_access_iterator_tag)
{
    ...
}
```

Версия для итераторов произвольного доступа, например, использует операции с произвольным доступом, а версия для двунаправленных итераторов нет. Благодаря иерархии дескрипторов итераторов (см. раздел 9.5) можно создать одну реализацию для итераторов нескольких категорий.

Реализация функции `distance()`

Примером выполнения описанной выше двухэтапной процедуры является реализация вспомогательной функции `distance()`, возвращающей расстояние между позициями двух итераторов и их элементов (см. раздел 9.3.3). Реализация итераторов произвольного доступа использует только операцию `-`.

Для всех других категорий итераторов функция возвращает количество инкрементов, необходимых для достижения конца диапазона.

```
// универсальная функция distance()

template <typename Iterator>
typename std::iterator_traits<Iterator>::difference_type
distance (Iterator pos1, Iterator pos2)
```

```
{
   return distance (pos1, pos2,
                    std::iterator_traits<Iterator>
                       ::iterator_category());
}

// функция distance() для итераторов произвольного доступа
template <typename RaIterator>
typename std::iterator_traits<RaIterator>::difference_type
distance (RaIterator pos1, RaIterator pos2,
          std::random_access_iterator_tag)
{
   return pos2 - pos1;
}

// функция distance() для итераторов ввода,
// одно- и двунаправленных итераторов
template <typename InIterator>
typename std::iterator_traits<InIterator>::difference_type
distance (InIterator pos1, InIterator pos2,
          std::input_iterator_tag)
{
   typename std::iterator_traits<InIterator>::difference_type d;
   for (d=0; pos1 != pos2; ++pos1, ++d) {
       ;
   }
   return d;
}
```

Здесь тип разности между итераторами используется как тип возвращаемого значения. Отметим, что вторая версия использует дескриптор для итераторов ввода, так что эта реализация может использоваться также одно- и двунаправленными итераторами, поскольку их дескрипторы являются производными от класса `input_iterator_tag`.

9.6. Создание пользовательских итераторов

Напишем собственный итератор. Как указано в предыдущем разделе, для этого нам нужны свойства этого итератора. Это можно сделать двумя способами.

1. Написать пять требуемых определений для обобщенной структуры `iterator_traits` (см. раздел 9.5).
2. Написать (частичную) специализацию структуры `iterator_traits`.

В первом случае стандартная библиотека C++ предоставляет специальный базовый класс `iterator<>`, содержащий определения типов. Нужно лишь передать эти типы.

```
class MyIterator
   : public std::iterator <std::bidirectional_iterator_tag,
                           type, std::ptrdiff_t, type*, type&> {
   ...
};
```

Первый шаблонный параметр определяет категорию итератора, второй — тип элементов *type*, третий — тип разности, четвертый определяет тип указателя, а пятый — тип ссылки. Последние три аргумента являются необязательными и имеют значения по умолчанию `ptrdiff_t`, *type** и *type*&. Таким образом, часто достаточно использовать следующее определение:

```
class MyIterator
   : public std::iterator <std::bidirectional_iterator_tag, type> {
     ...
};
```

Следующий пример демонстрирует, как написать пользовательский итератор. Это итератор вставки для ассоциативных и неупорядоченных массивов. В отличие от итераторов вставки из стандартной библиотеки C++ (см. раздел 9.4.2), в нем не используется позиция вставки:

Рассмотрим реализацию класса итератора:

```
// iter/assoiter.hpp

#include <iterator>

// шаблонный класс для итератора вставки
// для ассоциативных и неупорядоченных массивов

template <typename Container>
class asso_insert_iterator
: public std::iterator <std::output_iterator_tag,
                        typename Container::value_type>
{
   protected:
      Container& container; // контейнер, в который вставляются элементы
   public:
      // конструктор
     explicit asso_insert_iterator (Container& c) : container(c) {
}

   // оператор присваивания
   // - вставляет значение в контейнер
   asso_insert_iterator<Container>&
   operator= (const typename Container::value_type& value) {
      container.insert(value);
      return *this;
   }

   // разыменование - это фиктивная операция, возвращающая сам итератор
   asso_insert_iterator<Container>& operator* () {
      return *this;
   }

   // операция инкремента - это фиктивная операция,
   // возвращающая сам итератор
   asso_insert_iterator<Container>& operator++ () {
      return *this;
```

```
    }

    asso_insert_iterator<Container>& operator++ (int) {
        return *this;
    }
};

// вспомогательная функция для создания итератора вставки
template <typename Container>
inline asso_insert_iterator<Container> asso_inserter (Container& c)
{
    return asso_insert_iterator<Container>(c);
}
```

Класс `asso_insert_iterator` является производным от класса `iterator`, в котором определены соответствующие типы. Первый шаблонный аргумент — `output_iterator_tag`, который задает категорию итераторов. Второй аргумент — это тип значений, на которые ссылается итератор, т.е. тип `value_type` для контейнера. Поскольку итераторы вывода можно использовать только для записи, это определение типа не является необходимым, поэтому здесь можно передать `void`. Однако передача типа значений, показанная здесь, применима к любой категории итераторов.

В момент создания итератор хранит свой контейнер в переменной `container`, являющейся членом его класса. Любое присвоенное значение вставляется в контейнер с помощью функции `insert()`. Операторы `*` и `++` являются фиктивными и просто возвращают сам итератор. Таким образом, итератор поддерживает управление. Если используется обычный интерфейс итератора

```
*pos = value
```

то выражение `*pos` возвращает значение `*this`, которому присваивается новое значение. Это присваивание преобразовывается в вызов `insert(value)` для контейнера.

После определения класса итераторов вставки определяется обычная вспомогательная функция `asso_inserter` для создания и инициализации итератора вставки. В следующей программе такой итератор вставки используется для вставки элементов в неупорядоченное множество:

```
// iter/assoiter1.cpp

#include <iostream>
#include <unordered_set>
#include <vector>
#include <algorithm>
#include "print.hpp"
#include "assoiter.hpp"

int main()
{
    std::unordered_set<int> coll;
    // создаем итератор вставки для множества coll
    // - неудобный способ
    asso_insert_iterator<decltype(coll)> iter(coll);
```

```
    // вставляем элементы с обычным интерфейсом итераторов
    *iter = 1;
    iter++;
    *iter = 2;
    iter++;
    *iter = 3;
    PRINT_ELEMENTS(coll);

    // создаем итератор вставки для множества coll и вставляем элементы
    // - удобный способ
    asso_inserter(coll) = 44;
    asso_inserter(coll) = 55;
    PRINT_ELEMENTS(coll);

    // используем итераторы вставки и алгоритм
    std::vector<int> vals = { 33, 67, -4, 13, 5, 2 };
    std::copy (vals.begin(), vals.end(),            // источник
               asso_inserter(coll));                // получатель
    PRINT_ELEMENTS(coll);
}
```

Обычное применение класса `asso_inserter` демонстрирует вызов алгоритма `copy()`:

```
std::copy (vals.begin(), vals.end(),            // источник
           asso_inserter(coll));                // получатель
```

где выражение `asso_inserter(coll)` создает итератор вставки, вставляющий любой аргумент в множество `coll`, вызывая функцию

```
coll.insert(val).
```

Другие инструкции подробно демонстрируют работу итератора. Программа выводит следующие результаты:

```
1 2 3
55 44 1 2 3
-4 33 55 44 67 1 13 2 3 5
```

Отметим, что этот итератор можно использовать и с ассоциативными контейнерами. Таким образом, если в обеих директивах `include` и объявлении множества `coll` заменить `unordered_set` на `set`, то программа останется работоспособной (хотя элементы в контейнере будут упорядочены).

Глава 10

Функциональные объекты STL и лямбда-функции

В главе подробно рассматриваются механизмы передачи функциональности алгоритмам и функциям-членам — *функциональные объекты* (function objects), или *функторы* (functors), введенные в разделе 6.10. В ней описано полное множество функциональных объектов и адаптеров функций, а также привязки (связыватели — binders) и концепция функциональной композиции, приведены примеры пользовательских функциональных объектов и детали использования лямбда-функций (упоминаемых в разделах 3.1.10 и 6.9).

В заключение читатели узнают об устройстве и удивительном поведении алгоритмов `remove_if()` и `for_each()`.

10.1. Концепция функциональных объектов

Функциональный объект, или *функтор*, — это объект, имеющий определенный оператор `()`, так что в примере

```
Тип_функтора fo;
...
fo(...);
```

выражение `fo()` — это вызов оператора `()` для функционального объекта `fo`, а не вызов функции `fo()`. Функциональный объект можно рассматривать как обычную функцию, записанную необычным способом. Вместо того чтобы записать все операторы в теле функции:

```
void fo() {
   инструкции
}
```

мы записываем их в теле оператора `()` в классе функционального объекта:

```
class FunctionObjectType {
public:
   void operator() () {
     инструкции
   }
};
```

Несмотря на свою сложность, это определение имеет три важных преимущества.

1. Функциональный объект может быть более интеллектуальным, обладать большими возможностями, потому что у него есть состояние (ассоциированные члены, влияющие на поведение функционального объекта). Фактически вы можете иметь

два экземпляра одного и того же класса функционального объекта, которые могут одновременно иметь разные состояния. Обычные функции такими возможностями не обладают.

2. Каждый функциональный объект имеет свой тип. Следовательно, его тип можно передавать как шаблонный параметр, чтобы описать определенное поведение, и вы получите отличающиеся друг от друга контейнерные типы с разными функциональными объектами.
3. Функциональный объект обычно работает быстрее, чем указатель на функцию.

Более подробно эти преимущества объясняются в разделе 6.10.1, который содержит также пример, демонстрирующий, что функциональные объекты могут обладать более развитой логикой, чем обычные функции.

В следующих двух подразделах приводятся еще два примера, более подробно иллюстрирующие работу функциональных объектов. Первый пример показывает, как извлечь выгоду из того факта, что функциональный объект обычно имеет свой тип. Второй пример демонстрирует, как можно использовать состояние функционального объекта, и приводит к интересному свойству алгоритма `for_each()`, описанного в разделе 10.1.3.

10.1.1. Функциональные объекты как критерий сортировки

Программистам часто нужны упорядоченные коллекции элементов некоторого класса (например, коллекция объектов класса `Person`). Однако использовать для сортировки этих объектов оператор < иногда нежелательно, а иногда просто невозможно. Вместо этого желательно упорядочить эти объекты в соответствии с критерием сортировки, заданным некоторой функцией-членом. В этой ситуации может помочь функциональный объект. Рассмотрим следующий пример:

```
// fo/sort1.cpp

#include <iostream>
#include <string>
#include <set>
#include <algorithm>
using namespace std;

class Person {
   public:
      string firstname() const;
      string lastname() const;
      ...
};

// класс функционального предиката
// - операция () возвращает результат проверки того, что
// первый объект класса Person меньше второго объекта
class PersonSortCriterion {
   public:
      bool operator() (const Person& p1, const Person& p2) const {
         // объект класса Person меньше другого объекта класса Person
```

```
        // - если фамилия первого объекта меньше
        // - если фамилии совпадают, но имя первого объекта меньше
        return p1.lastname()<p2.lastname() ||
               (p1.lastname()==p2.lastname() &&
                p1.firstname()<p2.firstname());
    }
};
int main()
{
   // создаем множество со специальным критерием сортировки
   set<Person,PersonSortCriterion> coll;
   ...

   // что-то делаем с его элементами
   for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
      ...
   }
   ...
}
```

Множество `coll` использует специальный критерий сортировки `PersonSortCriterion`, определенный как класс функционального объекта. В классе `PersonSortCriterion` оператор `()` определен так, что он сравнивает два объекта класса `Person` по их фамилиям, а если они совпадают, по их именам. Конструктор множества `coll` автоматически создает экземпляр класса `PersonSortCriterion`, так что элементы упорядочиваются именно по этому критерию сортировки.

Отметим, что критерий сортировки `PersonSortCriterion` является *типом*. Следовательно, его можно использовать как шаблонный параметр множества. Это было бы невозможно, если бы мы реализовали критерий сортировки с помощью обычной функции (как показано в разделе 6.8.2).

Все множества с таким критерием сортировки имеют свой собственный тип. Их нельзя комбинировать или присваивать множествам, имеющим "обычный" или другой пользовательский критерий сортировки. Следовательно, ни одна операция множества не может повлиять на автоматический критерий сортировки; однако можно определить функциональные объекты, задающие разные критерии сортировки с одним и тем же типом (см. раздел 7.8.6).

10.1.2. Функциональные объекты, имеющие внутреннее состояние

Следующий пример демонстрирует, как можно использовать функциональные объекты в качестве функций, одновременно имеющих несколько состояний:

```
// fo/sequence1.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include <iterator>
#include "print.hpp"
using namespace std;
```

```
class IntSequence {
   private:
      int value;
   public:
      IntSequence (int initialValue) // конструктор
         : value(initialValue) {
   }

   int operator() () { // ''вызов функции''
       return value++;
   }
};

int main()
{
   list<int> coll;

   // вставляем значения от 1 до 9
   generate_n (back_inserter(coll), // старт
               9,                   // количество элементов
               IntSequence(1));     //генерирует значения, начиная с 1
   PRINT_ELEMENTS(coll);

   // заменяем элементы от второго до последнего
   // последовательностью элементов, начинающуюся с 42
   generate (next(coll.begin()), //старт
             prev(coll.end()),   // конец
             IntSequence(42));   //генерирует значения, начиная с 42

   PRINT_ELEMENTS(coll);
}
```

В этом примере функциональный объект `IntSequence` генерирует последовательность целых чисел. При каждом вызове оператор `()` возвращает текущее значение и выполняет его инкремент. Начальное значение можно задать с помощью аргумента конструктора.

Затем два таких функциональных объекта используются алгоритмами `generate()` и `generate_n()`, которые записывают сгенерированные значения в коллекцию: выражение `IntSequence(1)` в вызове

```
generate_n (back_inserter(coll),
            9,
            IntSequence(1));
```

создает такой функциональный объект, инициализированный значением `1`. Алгоритм `generate_n()` использует его девять раз для записи элемента, генерируя значения от `1` до `9`. Аналогично выражение `IntSequence(42)` генерирует последовательность чисел, начинающуюся с `42`. Алгоритм `generate()` заменяет элементы со второго и до последнего[1]. Результат работы программы выглядит следующим образом:

[1] Функции `std::next()` и `std::prev()` предусмотрены стандартом C++11 (см. раздел 9.3.2). Отметим, что выражения `++coll.begin()` и `--coll.end()` могут иногда не компилироваться (см. раздел 9.2.6)..

```
1 2 3 4 5 6 7 8 9
1 42 43 44 45 46 47 48 9
```

Используя другие варианты оператора `()`, можно легко создавать более сложные последовательности.

По умолчанию функциональные объекты передаются по значению, а не по ссылке. Следовательно, этот алгоритм не изменяет состояния функционального объекта. Например, следующий код дважды генерирует последовательность, начинающуюся с `1`:

```
IntSequence seq(1); // последовательность целых чисел, начинающаяся с 1

// вставляем последовательность, начинающуюся с 1
generate_n (back_inserter(coll), 9, seq);

// снова вставляем последовательность, начинающуюся с 1
generate_n (back_inserter(coll), 9, seq);
```

Передача функциональных объектов по значению, а не по ссылке позволяет передавать константные и временные выражения. В противном случае передача выражения `IntSequence(1)` была бы невозможной.

Недостатком передачи функционального объекта по значению является то, что мы не можем извлечь выгоду из модификаций состояния функциональных объектов. Алгоритмы могут модифицировать состояние функционального объекта, но получить доступ и обработать заключительное состояние невозможно, поскольку они создают внутренние копии функциональных объектов. В то же время доступ к заключительному состоянию может быть необходим, поэтому возникает вопрос, как получить "результат" работы алгоритма.

Существуют три способа получить результат или обратную связь между функциональными объектами и алгоритмами.

1. Можно хранить состояние во внешней памяти и позволить функциональному объекту обращаться к нему.
2. Можно передать функциональный объект по ссылке.
3. Можно использовать значение, возвращаемое алгоритмом `for_each()`.

Последняя возможность обсуждается в следующем подразделе.

Для передачи функционального объекта по ссылке нужно просто указать, что при вызове алгоритма типом функционального объекта является ссылка[2]. Рассмотрим пример:

```
// fo/sequence2.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include <iterator>
#include "print.hpp"
using namespace std;

class IntSequence {
  private:
    int value;
```

[2] Благодарю за это замечание Филипа Кёстера (Philip Köster)

```
   public:
      // конструктор
      IntSequence (int initialValue)
         : value(initialValue) {
      }

   // "вызов функции"
   int operator() () {
       return value++;
   }
};

int main()
{
   list<int> coll;

   IntSequence seq(1); // последовательность целых чисел,
                       // начинающаяся с 1

   // вставляем элементы от 1 до 4
   // - передаем функциональный объект по ссылке
   // чтобы он продолжил работу, начиная с 5
   generate_n<back_insert_iterator<list<int>>,
              int, IntSequence&>(back_inserter(coll), // начало
              4,                                      // количество элементов
              seq);                                   // генерирует значения
   PRINT_ELEMENTS(coll);

   // вставляем значения от 42 до 45
   generate_n (back_inserter(coll), // начало
               4,                   // количество элементов
               IntSequence(42));    // генерирует значения
   PRINT_ELEMENTS(coll);

   // продолжаем первую последовательность
   // - передаем функциональный объект по значению
   // чтобы он снова начал работу с 5
   generate_n (back_inserter(coll), // начало
               4,                   // количество элементов
               seq);                // генерирует значения
   PRINT_ELEMENTS(coll);

   // продолжаем первую последовательность
   generate_n (back_inserter(coll), // начало
               4,                   // количество элементов
               seq);                // генерирует значения
   PRINT_ELEMENTS(coll);
}
```

Программа выводит на экран следующие результаты:

```
1 2 3 4
1 2 3 4 42 43 44 45
```

```
1 2 3 4 42 43 44 45 5 6 7 8
1 2 3 4 42 43 44 45 5 6 7 8 5 6 7 8
```

При первом вызове алгоритма `generate_n()` функциональный объект `seq` передается по ссылке. Для этого явно указывается шаблонный параметр.

```
generate_n<back_insert_iterator<list<int>>,
           int, IntSequence&>(back_inserter(coll), // начало
                              4,                   // количество элементов
                              seq);                // генерирует значения
```

В результате внутреннее значение функционального объекта `seq` после вызова изменяется, и при втором использовании объекта `seq` (при третьем вызове) алгоритм `generate_n()` продолжает работу с последовательностью, полученной после первого вызова. Однако в этом вызове функциональный объект `seq` передается по значению.

```
generate_n (back_inserter(coll), // начало
            4,                   // количество элементов
            seq);                // количество значений
```

Следовательно, этот вызов не изменяет состояние функционального объекта `seq`. В результате последний вызов алгоритма `generate_n()` продолжает последовательность снова со значения 5.

10.1.3. Значение, возвращаемое алгоритмом `for_each()`

Передавать функциональный объект по ссылке, чтобы получить доступ к его заключительному состоянию, не обязательно, если использовать алгоритм `for_each()`. Этот алгоритм имеет уникальную возможность возвращать свой функциональный объект (ни один другой алгоритм делать это не может). Следовательно, можно запросить состояние вашего функционального объекта, проверив значение, возвращаемое алгоритмом `for_each()`.

Следующая программа представляет собой прекрасный пример использования значения, возвращаемого алгоритмом `for_each()`. Она демонстрирует процесс вычисления среднего значения последовательности:

```
// fo/foreach3.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// функциональный объект для вычисления среднего значения
class MeanValue {
  private:
    long num; // количество элементов
    long sum; // сумма всех значений
  public:
    // конструктор
    MeanValue () : num(0), sum(0) {
  }
```

```
    // "вызов функции"
    // - обработка нескольких элементов последовательности
    void operator() (int elem) {
        ++num;       // инкремент счетчика
        sum += elem; // добавляем значение
    }

    // возвращаем среднее значение
    double value () {
        return static_cast<double>(sum) / static_cast<double>(num);
    }
};

int main()
{
    vector<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8 };

    // вычисляем и выводим на экран среднее значение
    MeanValue mv = for_each (coll.begin(), coll.end(), // диапазон
                             MeanValue());             // операция
    cout << "mean value: " << mv.value() << endl;
}
```

Выражение `MeanValue()` создает функциональный объект, подсчитывающий количество элементов и вычисляющий сумму всех значений. При передаче этого функционального объекта в алгоритм `for_each()` он применяется к каждому элементу контейнера `coll`.

```
MeanValue mv = for_each (coll.begin(), coll.end(),
                         MeanValue());
```

Этот функциональный объект затем возвращается и присваивается объекту `mv`, поэтому мы можем запросить его состояние после выполнения этого оператора, вызвав функцию `mv.value()`. Таким образом, эта программа выводит на экран следующее значение:

```
mean value: 4.5
```

Класс `MeanValue` можно настроить более тонко, определив автоматическое преобразование типа в `double`. Затем можно непосредственно использовать среднее значение, вычисленное алгоритмом `for_each()`. Пример приведен в разделе 11.4.

Следует заметить, что лямбда-функции позволяют более удобно описать это поведение (соответствующий пример приведен в разделе 10.3.). Однако это не означает, что лямбда-функции всегда лучше функциональных объектов. Функциональные объекты являются более удобными в тех ситуациях, в которых требуется знать их тип, например, в объявлении хеш-функций, критерия сортировки, критерия эквивалентности или в неупорядоченных контейнерах. Тот факт, что функциональные объекты, как правило, вводятся глобально, позволяет включать их в заголовочные файлы и библиотеки, а лямбда-функции целесообразнее использовать для описания специфичного поведения, заданного локально.

10.1.4. Предикаты и функциональные объекты

Предикаты — это функции или функциональные объекты, возвращающие булево значение (значение, допускающее преобразование в тип `bool`). Однако не всякая функция,

возвращающая булево значение, является корректным предикатом для библиотеки STL. Это может привести к недоразумениям. Рассмотрим следующий пример:

```
// fo/removeif1.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

class Nth { // функциональный объект, возвращающий
            // значение true при n-м вызове
   private:
      int nth;   // вызов, для которого возвращается
                 // значение true
      int count; // счетчик вызовов
   public:
      Nth (int n) : nth(n), count(0) {
      }
   bool operator() (int) {
      return ++count == nth;
   }
};

int main()
{
   list<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   PRINT_ELEMENTS(coll,"coll: ");

   // удаляем третий элемент
   list<int>::iterator pos;
   pos = remove_if(coll.begin(),coll.end(), // диапазон
                   Nth(3));                 // критерий удаления
   coll.erase(pos,coll.end());
   PRINT_ELEMENTS(coll,"3rd removed: ");
}
```

В этой программе определяется функциональный объект `Nth`, возвращающий значение `true` при *n*-м вызове. Однако, если передать его алгоритму `remove_if()`, результат работы алгоритма, удаляющего все элементы, для которого унарный предикат возвращает значение `true` (см. раздел 11.7.1), окажется весьма странным[3].

```
coll: 1 2 3 4 5 6 7 8 9 10
3rd removed: 1 2 4 5 7 8 9 10
```

Удалены два элемента — третий и шестой. Это произошло потому, что обычная реализация алгоритма автоматически копирует предикат во время работы алгоритма.

```
template <typename ForwIter, typename Predicate>
ForwIter std::remove_if(ForwIter beg, ForwIter end,
```

[3] По крайней мере так получается при работе с компиляторами gcc 4.5 и Visual C++ 2010, но другие платформы могут привести к другим результатам; этот вопрос обсуждается ниже.

```
                        Predicate op)
{
   beg = find_if(beg, end, op);
   if (beg == end) {
       return beg;
   }
   else {
       ForwIter next = beg;
       return remove_copy_if(++next, end, beg, op);
   }
}
```

Этот алгоритм использует алгоритм `find_if()` для поиска первого элемента, подлежащего удалению. Однако затем для обработки оставшихся элементов, если они есть, алгоритм использует копию переданного предиката `op`. В этом месте функциональный объект `Nth` используется в его исходном состоянии, и алгоритм снова удаляет третий среди оставшихся элементов, который фактически является седьмым.

Это — не ошибка. Стандарт не регламентирует, как часто предикат может копироваться в алгоритме. Следовательно, для того чтобы поведение стандартной библиотеки C++ было гарантированным, не следует передавать функциональный объект, поведение которого зависит от того, сколько раз он копируется или вызывается. Таким образом, если вызвать унарный предикат два раза с двумя одинаковыми аргументами, то оба раза предикат обязан вернуть один и тот же результат.

Другими словами: **предикат не должен иметь состояния**. Иначе говоря, предикат не должен изменять свое состояние вследствие вызова, а копия предиката должна иметь такое же состояние, что и оригинал. Для того чтобы запретить изменение состояния предиката во время вызова функции, оператор `()` необходимо определить как константную функцию-член.

Описанное выше неожиданное поведение можно предотвратить и гарантировать, что алгоритм будет работать как надо, даже с таким функциональным объектом, как `Nth`, не снижая быстродействия. Можно реализовать алгоритм `remove_if()` так, чтобы вызов алгоритма `find_if()` заменялся его содержимым.

```
template <typename ForwIter, typename Predicate>
ForwIter std::remove_if(ForwIter beg, ForwIter end,
                        Predicate op)
{
   while (beg != end && !op(*beg)) {
       ++beg;
   }

   if (beg == end) {
      return beg;
   }
   else {
      ForwIter next = beg;
      return remove_copy_if(++next, end, beg, op);
   }
}
```

Итак, было бы целесообразно заменить реализацию алгоритма `remove_if()` или сообщить разработчикам библиотеки о необходимости сделать это. Насколько я знаю, эта проблема в действующей реализации библиотеки возникла только при работе с алгоритмом `remove_if()`. Алгоритм `remove_copy_if()` работает так, как ожидается. (Вопрос, следует ли стандартной библиотеке C++ гарантировать ожидаемое поведение в таких ситуациях, как в предыдущем примере, все еще остается предметом обсуждений.) Однако для обеспечения переносимости программ никогда не следует полагаться на детали реализации. Оператор вызова предиката всегда следует объявлять как константную функцию-член.

При работе с лямбда-функциями состояние можно сделать общим для всех копий функционального объекта, так что эта проблема не возникает. Подробности описаны в разделе 10.3.2.

10.2. Стандартные функциональные объекты и привязки

Как указано в разделе 6.10.2, стандартная библиотека C++ содержит много стандартных функциональных объектов и привязок, позволяющих создавать более сложные функциональные объекты.

Эта возможность называется *функциональной композицией* (functional composition). Она основана на фундаментальных функциональных объектах и адаптерах, описанных в этом разделе. Для использования этих функциональных объектов и привязок в программу следует включить заголовочный файл `<functional>`.

```
#include <functional>
```

10.2.1. Стандартные функциональные объекты

Все предопределенные функциональные объекты перечислены в табл. 10.1 (функциональные объекты `bit_and`, `bit_or` и `bit_xor` предусмотрены стандартом C++11).

Таблица 10.1. Стандартные функциональные объекты

Выражение	Результат
`negate<`*type*`>()`	*- param*
`plus<`*type*`>()`	*param1 + param2*
`minus<`*type*`>()`	*param1 - param2*
`multiplies<`*type*`>()`	*param1 * param2*
`divides<`*type*`>()`	*param1 / param2*
`modulus<`*type*`>()`	*param1 % param2*
`equal_to<`*type*`>()`	*param1 == param2*
`not_equal_to<`*type*`>()`	*param1 != param2*
`less<`*type*`>()`	*param1 < param2*
`greater<`*type*`>()`	*param1 > param2*

Окончание табл. 10.1

Выражение	Результат
`less_equal<type>()`	*param1* <= *param2*
`greater_equal<type>()`	*param1* >= *param2*
`logical_not<type>()`	*!param*
`logical_and<type>()`	*param1* && *param2*
`logical_or<type>()`	*param1* || *param2*
`bit_and<type>()`	*param1* & *param2*
`bit_or<type>()`	*param1* | *param2*
`bit_xor<type>()`	*param1* ^ *param2*

Функциональный объект `less<>` — это критерий, заданный по умолчанию и предназначенный для упорядочения и сравнения объектов с помощью функций сортировки и ассоциативных контейнеров. Таким образом, операции сортировки, заданные по умолчанию, всегда создают возрастающий порядок (*element* < *nextElement*). Функциональный объект `equal_to<>` — это критерий эквивалентности по умолчанию для неупорядоченных контейнеров.

Для сравнения строк на разных языках стандартная библиотека C++ позволяет задавать функциональные объекты локализации в качестве критерия сортировки строк (см. раздел 16.3).

10.2.2. Функциональные адаптеры и привязки

Функциональный адаптер — это функциональный объект, предназначенный для создания композиций, состоящих из функциональных объектов, определенных значений и специальных функций (соответствует шаблону *Компоновщик* в книге [*GoF:DesignPatterns*]). Однако со временем способ создания композиций из функциональных объектов изменился. Фактически все такие свойства, предусмотренные в стандарте C++98, объявлены устаревшими в стандарте C++11, в котором описаны более удобные и гибкие адаптеры. Сначала будет представлен современный способ создания композиций из функциональных объектов, а в разделе 10.2.4 будет дан более короткий обзор устаревших свойств.

В табл. 10.2 перечислены функциональные адаптеры, предусмотренные в стандартной библиотеке C++ в соответствии со стандартом C++11.

Таблица 10.2. Стандартные функциональные адаптеры

Выражение	Результат
`bind(`*op*`,`*args*`...)`	Связывает списком аргументов *args* с операцией *op*
`mem_fn(`*op*`)`	Вызывает операцию *op()* как функцию-член объекта или объекта, на который ссылается указатель
`not1(`*op*`)`	Унарное отрицание: !*op*(*param*)
`not2(`*op*`)`	Бинарное отрицание: !op(*param1*,*param2*)

Наиболее важным адаптером является `bind()`, он позволяет делать следующее:

- адаптировать и создавать новые функциональные объекты из существующих или стандартных функциональных объектов;
- вызывать глобальные функции;
- вызывать функции-члены объектов, указателей на объекты и интеллектуальных указателей на объекты.

Адаптер `bind()`

В принципе адаптер `bind()` связывает параметры *вызываемых объектов* (см. раздел 4.4). Следовательно, если функция, функция-член, функциональный объект или лямбда-функция требует каких-то параметров, их можно связать с конкретными или передаваемыми аргументами. Конкретные аргументы можно просто назвать. Для передаваемых аргументов можно использовать стандартные *заполнители* `_1`, `_2`, ..., определенные в пространстве имен `std::placeholders`.

Как правило, привязки используются для указания параметров при использовании стандартных функциональных объектов, предусмотренных в стандартной библиотеке C++ (см. раздел 10.2.1). Рассмотрим пример:

```
// fo/bind1.cpp

#include <functional>
#include <iostream>

int main()
{
   auto plus10 = std::bind(std::plus<int>(),
                           std::placeholders::_1,
                           10);
   std::cout << "+10: " << plus10(7) << std::endl;

   auto plus10times2 = std::bind(std::multiplies<int>(),
                                 std::bind(std::plus<int>(),
                                           std::placeholders::_1,
                                           10),
                                 2);
   std::cout << "+10 *2: " << plus10times2(7) << std::endl;

   auto pow3 = std::bind(std::multiplies<int>(),
                         std::bind(std::multiplies<int>(),
                                   std::placeholders::_1,
                                   std::placeholders::_1),
                         std::placeholders::_1);
   std::cout << "x*x*x: " << pow3(7) << std::endl;

   auto inversDivide = std::bind(std::divides<double>(),
                                 std::placeholders::_2,
                                 std::placeholders::_1);
   std::cout << "invdiv: " << inversDivide(49,7) << std::endl;
}
```

Здесь определены четыре разных привязки, представляющие собой функциональные объекты. Например, привязка `plus10`, определенная как

```
std::bind(std::plus<int>(),
          std::placeholders::_1,
          10)
```

представляет собой функциональный объект, который автоматически вызывает функциональный объект `plus<>` (т.е. операцию +), передавая ему заполнитель `_1` в качестве первого параметра/операнда и `10` в качестве второго параметра/операнда. Заполнитель `_1` представляет первый аргумент, передаваемый выражению, как единое целое. Следовательно, для любого аргумента, передаваемого в это выражение, данный функциональный объект будет возвращать значение этого аргумента, увеличенное на `10`.

Для того чтобы избежать утомительного повторения пространства имен `placeholders`, можно использовать соответствующую директиву `using`. Таким образом, две директивы `using` позволяют сделать оператор более компактным.

```
using namespace std;
using namespace std::placeholders;
bind (plus<int>(), _1, 10) // param1+10
```

Привязку также можно вызывать непосредственно. Например, выражение

```
std::cout << std::bind(std::plus<int>(),_1,10)(32) << std::endl;
```

запишет `42` в стандартный поток вывода, если передать этот функциональный объект алгоритму, и алгоритм может применить его к каждому элементу, с которым он работает. Рассмотрим пример:

```
// добавляем 10 к каждому элементу
std::transform (coll.begin(), coll.end(),           // источник
                coll.begin(),                       // получатель
                std::bind(std::plus<int>(),_1,10)); // операция
```

Аналогично можно определить привязку, представляющую критерий сортировки. Например, для поиска первого элемента, превышающего `42`, можно связать функциональный объект `greater<>`, переданный в качестве первого аргумента, с числом `42`, указанным в качестве второго аргумента.

```
// ищем первый элемент > 42
auto pos = std::find_if (coll.begin(),coll.end(),
                         std::bind(std::greater<int>(),_1,42))
```

Отметим, что для используемого стандартного функционального объекта всегда необходимо задавать тип аргумента. Если типы не совпадают, выполняется преобразование или возникает ошибка компиляции.

Остальные инструкции в этой программе показывают, что привязки могут быть вложенными и создавать еще более сложные функциональные объекты. Например, следующее выражение определяет функциональный объект, добавляющий число `10` к передаваемому аргументу, а затем умножающий его на `2` (пространство имен опущено).

```
bind(multiplies<int>(), // (param1+10)*2
     bind(plus<int>(),_1,
```

```
                10),
    2);
```

Как видим, выражения вычисляются в направлении изнутри наружу.

Аналогично можно возвести значение в куб, объединив два объекта `multiplies<>` с тремя заполнителями передаваемого аргумента.

```
bind(multiplies<int>(), // (param1*param1)*param1
     bind(multiplies<int>(),_1,
                            _1),
     _1);
```

Последнее выражение определяет функциональный объект, в котором аргументы для деления переставлены местами. Таким образом, он делит второй аргумент на первый.

```
bind(divides<double>(),_2, // param2/param1
                       _1);
```

Итак, программа выводит на экран следующие результаты:

```
+10: 17
+10 *2: 34
x*x*x: 343
invdiv: 0.142857
```

Другие примеры использования привязок приведены в разделе 6.10.3. В разделе 10.3.1 показано, как то же самое можно сделать с помощью лямбда-функций.

Вызов глобальных функций

Следующий пример демонстрирует, как можно использовать привязку `bind()` для вызова глобальных функций (вариант с использованием лямбда-функций приведен в разделе 10.3.3).

```
// fo/compose3.cpp

#include <iostream>
#include <algorithm>
#include <functional>
#include <locale>
#include <string>
using namespace std;
using namespace std::placeholders;

char myToupper (char c)
{
   std::locale loc;
   return std::use_facet<std::ctype<char> >(loc).toupper(c);
}

int main()
{
   string s("Internationalization");
   string sub("Nation");
```

```
    // поиск подстроки без учета регистра
    string::iterator pos;
    pos = search (s.begin(),s.end(),      // строка, в которой идет поиск
                  sub.begin(),sub.end(),  // искомая подстрока
                  bind(equal_to<char>(),  // критерий сравнения
                       bind(myToupper,_1),
                       bind(myToupper,_2)));
    if (pos != s.end()) {
       cout << "\"" << sub << "\" is part of \"" << s << "\""
            << endl;
    }
}
```

Здесь алгоритм `search()` использован для проверки, является ли строка `sub` подстрокой строки `s` без учета регистра. С помощью привязки

```
bind(equal_to<char>(),
     bind(myToupper,_1),
     bind(myToupper,_2)));
```

создается вызов функционального объекта:

```
myToupper(param1)==myToupper(param2)
```

в котором `myToupper()` — пользовательская вспомогательная функция для преобразования строки в верхний регистр (см. раздел 16.4.4).

Отметим, что привязка `bind()` автоматически копирует передаваемые аргументы. Для того чтобы функциональный объект использовал ссылки на передаваемый аргумент, следует использовать функции `ref()` или `cref()` (см. раздел 5.4.3). Рассмотрим пример:

```
void incr (int& i)
{
   ++i;
}

int i=0;
bind(incr,i)();       // инкремент копии i, на i не влияет
bind(incr,ref(i))();  // инкремент i
```

Вызов функций-членов

Следующая программа демонстрирует, как привязка `bind()` позволяет вызвать функцию-член (вариант с использованием лямбда-функций приведен в разделе 10.3.3).

```
// fo/bind2.cpp

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

```
using namespace std;
using namespace std::placeholders;

class Person {
   private:
      string name;
   public:
      Person (const string& n) : name(n) {
      }
      void print () const {
         cout << name << endl;
      }

      void print2 (const string& prefix) const {
         cout << prefix << name << endl;
      }
   ...
};

int main()
{
   vector<Person> coll
         = { Person("Tick"), Person("Trick"), Person("Track") };


   // вызываем функцию-член print() каждого объекта класса Person
   for_each (coll.begin(), coll.end(),
             bind(&Person::print,_1));
   cout << endl;

   // вызываем функцию-член print2() каждого объекта класса Person
   // с дополнительным аргументом
   for_each (coll.begin(), coll.end(),
             bind(&Person::print2,_1,"Person: "));
   cout << endl;

   // вызываем функцию-член print2() временного объекта класса Person
   bind(&Person::print2,_1,"This is: ")(Person("nico"));
}
```

Здесь привязка

```
bind(&Person::print,_1)
```

определяет функциональный объект, вызывающий функцию-член *param1*`.print()` переданного объекта класса `Person`. Иначе говоря, поскольку первый аргумент является функцией-членом, второй аргумент определяет объект, для которого эта функция вызывается.

Этой функции-члену передаются все дополнительные аргументы. Это значит, что привязка

```
bind(&Person::print2,_1,"Person: ")
```

определяет функциональный объект, вызывающий функцию-член *param1*`.print2 ("Person: ")` для любого переданного объекта класса `Person`.

Здесь передаваемые объекты являются членами вектора `coll`, но в принципе объекты можно передавать непосредственно. Например:

```
Person n("nico");
bind(&Person::print2,_1,"This is: ")(n);
```

вызывает `n.print2("This is: ")`.

Результаты работы этой программы имеют следующий вид:

```
Tick
Trick
Track
Person: Tick
Person: Trick
Person: Track
This is: nico
```

Отметим, что привязке `bind()` можно передавать также указатели на объекты и даже интеллектуальные указатели.

```
std::vector<Person*> cp;
...
std::for_each (cp.begin(), cp.end(),
               std::bind(&Person::print,
                         std::placeholders::_1));

std::vector<std::shared_ptr<Person>> sp;
...
std::for_each (sp.begin(), sp.end(),
               std::bind(&Person::print,
                         std::placeholders::_1));
```

Отметим, что с помощью привязки можно также вызывать модифицирующие функции-члены.

```
class Person {
   public:
      ...
      void setName (const std::string& n) {
          name = n;
      }
};

std::vector<Person> coll;
...
std::for_each (coll.begin(), coll.end(), // присваиваем всем одинаковые имена
               std::bind(&Person::setName,
                         std::placeholders::_1,
                         "Paul"));
```

Возможен также вызов виртуальных функций-членов. Если связывается метод базового класса, а объект принадлежит производному классу, то будет вызвана правильная виртуальная функция производного класса.

Адаптер mem_fn()

Для вызова функций-членов можно также использовать адаптер mem_fn(), в котором можно не указывать заполнитель для объекта, для которого вызывается функция-член.

```
std::for_each (coll.begin(), coll.end(),
               std::mem_fn(&Person::print));
```

Таким образом, оператор () из объекта, возвращенного адаптером mem_fn(), просто вызывает инициализированную им функцию-член. Вызов функции относится к объекту, который передается как первый аргумент, а дополнительные аргументы передаются функции-члену в качестве параметров.

```
std::mem_fn(&Person::print)(n);              // вызывает n.print()

std::mem_fn(&Person::print2)(n,"Person: "); // вызывает n.print2("Person: ")
```

Однако для связывания дополнительного аргумента с функциональным объектом снова необходимо использовать привязку bind().

```
std::for_each (coll.begin(), coll.end(),
               std::bind(std::mem_fn(&Person::print2),
                         std::placeholders::_1,
                         "Person: "));
```

Связывание с данными-членами

Привязку можно установить и к данным-членам. Рассмотрим следующий пример (пространство имен опущено):[4]

```
map<string,int> coll; // отображение целых чисел, ассоциированных со строками
...
// накапливаем все значения (вторые члены элементов)
int sum
  = accumulate (coll.begin(), coll.end(),
                0,
                bind(plus<int>(),
                     _1,
                     bind(&map<string,int>::value_type::second,
                          _2)));
```

Здесь вызывается алгоритм accumulate(), использующий бинарный предикат для суммирования всех значений всех элементов (см. раздел 11.11.1). Однако, поскольку мы используем отображение, в котором элементами являются пары "ключ–значение", для доступа к значению элемента мы с помощью привязки

```
bind(&map<string,int>::value_type::second,_2)
```

связываем второй аргумент каждого вызова предиката с его членом second.

[4] Этот пример основан на программе, позаимствованной из книги [*Karlsson:Boost*] с любезного разрешения автора.

Адаптеры not1() и not2()

Адаптеры `not1()` и `not2()` можно считать почти устаревшими[5]. Их единственное предназначение — отрицание стандартных функциональных объектов. Например:

```
std::sort (coll.begin(), coll.end(),
           std::not2(std::less<int>()));
```

Этот код выглядит более удобным, чем

```
std::sort (coll.begin(), coll.end(),
           std::bind(std::logical_not<bool>(),
                     std::bind(std::less<int>(),_1,_2)));
```

Однако для использования адаптеров `not1()` и `not2()` невозможно написать реальный сценарий, потому что здесь можно просто использовать другой стандартный функциональный объект.

```
std::sort (coll.begin(), coll.end(),
           std::greater_equal<int>());
```

Более важным является то, что вызов адаптера `not2()` для функционального объекта `less<>` в любом случае ошибочный. Вероятно, программист имел в виду изменение направления сортировки с возрастающей на убывающую. Но отрицание оператора < — это оператор >=, а не >. Фактически функциональный объект `greater_equal<>` приводит к непредсказуемым последствиям, потому что алгоритм `sort()` требует *строгого квазиупорядочения*, который обеспечивается операцией <, но не операцией >=, нарушающей требование антисимметричности (см. раздел 7.7). Таким образом, можно либо передать функциональный объект

```
greater<int>()
```

либо поменять порядок аргументов

```
bind(less<int>(),_2,_1)
```

Другие примеры использования адаптеров `not1()` и `not2()` с другими устаревшими функциональными адаптерами приведены в разделе 10.2.4.

10.2.3. Пользовательские функциональные объекты для функциональных адаптеров

Привязки можно использовать и с пользовательскими функциональными объектами. Следующий пример демонстрирует полное описание функционального объекта, возводящего первый аргумент в степень, равную значению второго аргумента.

```
// fo/fopow.hpp

#include <cmath>
template <typename T1, typename T2>
```

[5] Действительно, они были близки к объявлению устаревшими уже в стандарте C++11 [*N3198:DeprAdapt*].

```
struct fopow
{
   T1 operator() (T1 base, T2 exp) const {
      return std::pow(base,exp);
   }
};
```

Отметим, что первый аргумент и возвращаемое значение имеют одинаковый тип `T1`, а показатель степени имеет другой тип, `T2`.

Следующая программа демонстрирует применение пользовательского функционального объекта `fopow<>()`. В частности, она использует функциональные адаптеры `fopow<>()` и `bind()`.

```
// fo/fopow1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <functional>
#include "fopow.hpp"
using namespace std;
using namespace std::placeholders;

int main()
{
   vector<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

   // выводим на экран число 3, возведенное в степени,
   // равные значениям всех элементов

   transform (coll.begin(), coll.end(),            // источник
              ostream_iterator<float>(cout," "),   // получатель
              bind(fopow<float,int>(),3,_1));      // операция
   cout << endl;

   // выводим на экран все элементы, возведенные в степень 3
   transform (coll.begin(), coll.end(),            // источник
              ostream_iterator<float>(cout," "),   // получатель
              bind(fopow<float,int>(),_1,3));      // операция
   cout << endl;
}
```

Программа выводит на экран следующие числа:

```
3 9 27 81 243 729 2187 6561 19683
1 8 27 64 125 216 343 512 729
```

Следует подчеркнуть, что функциональный объект `fopow<>()` реализован для типов `float` и `int`. Если число типа `int` используется и как основание, и как показатель степени, можно вызвать функцию `pow()` с двумя аргументами типа `int`, но этот вариант не является переносимым, поскольку по стандарту функция `pow()` перегружена для нескольких, но не для всех элементарных типов.

```
transform (coll.begin(), coll.end(),
           ostream_iterator<int>(cout," "),
           bind1st(fopow<int,int>(),3));     // ОШИБКА: неоднозначность
```

Подробное описание этой проблемы приведено в разделе 17.3.

10.2.4. Устаревшие функциональные адаптеры

В табл. 10.3 перечислены классы стандартных функциональных адаптеров, существовавших в стандартной библиотеке C++ до появления стандарта C++11 и в настоящее время устаревших[6]. Ниже будут приведены примеры их использования.

Таблица 10.3. Устаревшие стандартные функциональные адаптеры

Выражение	Результат
`bind1st(`*op*`,`*arg*`)`	Вызывает операцию *op*(*arg*,*param*)
`bind2nd(`*op*`,`*arg*`)`	Вызывает операцию *op*(*param*,*arg*)
`ptr_fun(`*op*`)`	Вызывает операцию **op*(*param*) или **op*(*param1*,*param2*)
`mem_fun(`*op*`)`	Вызывает операцию *op*() как функцию-член объекта, на который ссылается указатель
`mem_fun_ref(`*op*`)`	Вызывает операцию *op*() как функцию-член объекта
`not1(`*op*`)`	Унарное отрицание: !*op*(*param*)
`not2(`*op*`)`	Бинарное отрицание: !*op*(*param1*,*param2*)

Эти адаптеры требуют определения некоторых типов в используемых функциональных объектах. Для определения этих типов в стандартной библиотеке C++ предусмотрены специальные базовые классы для функциональных адаптеров: `std::unary_function<>` и `std::binary_function<>`. В настоящее время эти классы также устарели.

Адаптеры `bind1st()` и `bind2nd()`, как и `bind()`, связывают параметр с фиксированными позициями. Например:

```
// найти первый элемент, превышающий 42
std::find_if (coll.begin(),coll.end(),                  // диапазон
              std::bind2nd(std::greater<int>(),42)) // критерий
```

Однако адаптеры `bind1st()` и `bind2nd()` нельзя использовать для создания композиции привязок или для непосредственной передачи функций.

Адаптеры `not1()` и `not2()` "почти устарели", потому что они полезны только при работе с другими устаревшими функциональными адаптерами. Например, алгоритм

```
std::find_if (coll.begin(), coll.end(),
              std::not1(std::bind2nd(std::modulus<int>(),2)));
```

находит позицию первого четного целого числа (операция `%2` возвращает `0` для всех четных значений, а адаптер `not1()` отрицает `0`, возвращая значение `true`). Однако это можно сделать более удобно с помощью новых привязок.

[6] Кроме адаптеров `not1()` и `not2()`, которые не были официально объявлены устаревшими, в реальных программах встречаются и другие устаревшие функциональные адаптеры.

```
std::find_if (coll.begin(), coll.end(),
              std::bind(std::logical_not<bool>(),
                        std::bind(std::modulus<int>(),
                                  std::placeholders::_1,
                                  2)));
```

Еще лучше применить в этой ситуации лямбда-функцию.

```
std::find_if (coll.begin(), coll.end(),
              [](int elem){
                  return elem%2==0;
              });
```

Адаптер `ptr_fun()` позволяет вызывать обычные функции. Например, допустим, что у нас есть глобальная функция, проверяющая некое условие для каждого параметра.

```
bool check(int elem);
```

Для того чтобы найти первый элемент, не удовлетворяющий проверяемому условию, можно выполнить оператор

```
std::find_if (coll.begin(), coll.end(),        // диапазон
              std::not1(std::ptr_fun(check))); // критерий поиска
```

Вторая форма адаптера `ptr_fun()` используется тогда, когда у нас есть глобальная функция с двумя параметрами, которую мы хотим использовать как унарную функцию.

```
// найти первую непустую строку
std::find_if (coll.begin(), coll.end(),
              std::bind2nd(std::ptr_fun(std::strcmp),""));
```

Здесь С-функция `strcmp()` используется для сравнения каждого элемента с пустой С-строкой. Если обе строки совпадают, то функция `strcmp()` возвращает `0`, что эквивалентно значению `false`. Итак, этот вызов алгоритма `find_if()` возвращает позицию первого элемента, не являющегося пустой строкой.

Адаптеры `mem_fun()` и `mem_fun_ref()` позволяют определить функциональные объекты, вызывающие функции-члены[7]. Например:

```
class Person {
  public:
    void print () const;
    ...
};

const std::vector<Person> coll;
...
// вызываем функцию-член print() из каждого объекта класса Person
std::for_each (coll.begin(), coll.end(),
               std::mem_fun_ref(&Person::print));
```

[7] Эти адаптеры функций-членов используют вспомогательные классы `mem_fun_t`, `mem_fun_ref_t`, `const_mem_fun_t`, `const_mem_fun_ref_t`, `mem_fun1_t`, `mem_fun1_ref_t`, `const_mem_fun1_t` и `const_mem_fun1_ref_t`.

Отметим, что функции-члены, вызываемые адаптерами `mem_fun_ref()` и `mem_fun()` и передаваемые как аргументы адаптерам `bind1st()` или `bind2nd()`, должны быть *константными*.

10.3. Использование лямбда-функций

Как указано в разделе 3.1.10, в стандарте C++11 появились лямбда-функции. Они представляют собой мощный и очень удобный механизм локальной функциональности, особенно полезный для уточнения деталей алгоритмов и функций-членов. Хотя лямбда-функции являются свойствами языка, их использование настолько важно для стандартной библиотеки C++, что я решил описать их здесь.

В разделе 6.9 отмечено, что лямбда-функции являются значительным улучшением языка C++ при работе с библиотекой STL, потому что теперь у нас есть интуитивный и понятный способ передачи индивидуального поведения алгоритмам и функциям-членам контейнеров. Если алгоритму требуется передать индивидуальное поведение, достаточно описать его как любую другую функцию именно в том месте, где он требуется.

Лучше всего продемонстрировать использование лямбда-функций на примере, сравнив его с соответствующим кодом, в котором не используются лямбда-функции. В следующих подразделах приведено несколько примеров функциональности, для достижения которой ранее использовались функциональные объекты и адаптеры, такие как `bind()`.

10.3.1. Лямбда-функции и адаптеры

Возьмем, например, файл `fo/bind1.cpp`, представленный в разделе 10.2.2. При использовании лямбда-функций соответствующий код выглядит следующим образом:

```
// fo/lambda1.cpp

#include <iostream>
int main()
{
   auto plus10 = [] (int i) {
                    return i+10;
                 };

   std::cout << "+10:     " << plus10(7) << std::endl;

   auto plus10times2 = [] (int i) {
                         return (i+10)*2;
                       };

   std::cout << "+10 *2: " << plus10times2(7) << std::endl;

   auto pow3 = [] (int i) {
                  return i*i*i;
               };

   std::cout << "x*x*x: " << pow3(7) << std::endl;
```

```
    auto inversDivide = [] (double d1, double d2) {
                            return d2/d1;
                        };

    std::cout << "invdiv: " << inversDivide(49,7) << std::endl;
}
```

Сравните это с объявлением функционального объекта. Объявление "добавить 10 и умножить на 2" с помощью привязок выглядит следующим образом:

```
auto plus10times2 = std::bind(std::multiplies<int>(),
                              std::bind(std::plus<int>(),
                                        std::placeholders::_1,
                                        10),
                              2);
```

Та же самая функциональная возможность, определенная с помощью лямбда-функции, выглядит так:

```
auto plus10times2 = [] (int i) {
                        return (i+10)*2;
                    };
```

10.3.2. Лямбда-функции и функциональные объекты, имеющие состояние

Теперь заменим пользовательский функциональный объект лямбда-функцией. Рассмотрим пример с вычислением среднего значения элементов из раздела 10.1.3. Версия с лямбда-функциями приведена ниже.

```
// fo/lambda2.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8 };

    // вычисляем и выводим среднее значение
    long sum = 0;
    for_each (coll.begin(), coll.end(), // диапазон
              [&sum] (int elem) {
                  sum += elem;
              });
    double mv = static_cast<double>(sum)/static_cast<double>(coll.size());
    cout << "mean value: " << mv << endl;
}
```

Здесь вместо определения класса для передаваемого функционального объекта мы просто передаем требуемую функциональность. Однако результат вычислений хранится вне лямбда-функции в переменной sum, так что для вычисления среднего значения нам потребуется именно эта переменная.

При работе с функциональным объектом это состояние (переменная sum) полностью инкапсулируется, и мы можем предоставить для работы с этим состоянием дополнительные функции-члены (например, для получения среднего значения из переменной sum).

```
MeanValue mv = for_each (coll.begin(), coll.end(), //  диапазон
                         MeanValue());             //  операция
cout << "mean value: " << mv.value() << endl;
```

Так что с точки зрения вызовов пользовательский функциональный объект можно считать более компактным и менее подверженным ошибкам, чем лямбда-функции.

При работе с состоянием необходимо аккуратно использовать ключевое слово mutable. Рассмотрим пример из раздела 10.1.4, в котором критерий поиска третьего элемента задан функциональным объектом, имеющим состояние. Соответствующая версия, написанная с помощью лямбда-функций, строго говоря, должна передавать внутренний счетчик, выражающий ее состояние, по значению, потому что вне вызываемого алгоритма он не нужен. Используя ключевое слово mutable, можно обеспечить всем "вызовам функции" доступ для записи к этому состоянию.

```
// fo/lambda3.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
   list<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   PRINT_ELEMENTS(coll,"coll:        ");

   // удаляем третий элемент
   list<int>::iterator pos;
   int count=0;                                  // счетчик вызовов
   pos = remove_if(coll.begin(),coll.end(), // диапазон
                   [count] (int) mutable {  // критерий удаления
                        return ++count == 3;
                   });
   coll.erase(pos,coll.end());

   PRINT_ELEMENTS(coll,"3rd removed: ");
}
```

Однако, как указано в разделе 10.1.4, может возникнуть проблема с удалением третьего и шестого элементов. В итоге мы получим следующий результат:

```
coll: 1 2 3 4 5 6 7 8 9 10
3rd removed: 1 2 4 5 7 8 9 10
```

Как и ранее, причина ошибки заключается в том, что лямбда-объект копируется алгоритмом `remove_if()` во время его выполнения, так что существуют два лямбда-объекта, удаляющих третий элемент. Следовательно, состояние дублируется.

Если передать аргумент по ссылке и не использовать ключевое слово `mutable`, то все будет работать, как ожидается, потому что оба лямбда-объекта, используемые алгоритмом `remove_if()`, разделяют одно и то же состояние. Таким образом, код

```
int count=0;                                  // счетчик вызовов
pos = remove_if(coll.begin(),coll.end(),  // диапазон
                [&count] (int) {              // критерий удаления
                     return ++count == 3;
     });
```

выводит следующие результаты:

```
coll: 1 2 3 4 5 6 7 8 9 10
3rd removed: 1 2 4 5 6 7 8 9 10
```

10.3.3. Лямбда-функции, вызывающие глобальные функции и функции-члены

Разумеется, любая лямбда-функция может вызвать другую функцию, так что лямбда-версия программы `fo/compose3.cpp` из раздела 10.2.2 принимает следующий вид:

```
// fo/lambda4.cpp

#include <iostream>
#include <algorithm>
#include <locale>
#include <string>
using namespace std;

char myToupper (char c)
{
   std::locale loc;
   return std::use_facet<std::ctype<char> >(loc).toupper(c);
}

int main()
{
   string s("Internationalization");
   string sub("Nation");

   // поиск подстроки независимо от регистра
   string::iterator pos;
   pos = search (s.begin(),s.end(),      // строка, в которой идет поиск
                 sub.begin(),sub.end(),  // искомая подстрока
                 [] (char c1, char c2) {  // критерия сравнения
                     return myToupper(c1)==myToupper(c2);
   });
   if (pos != s.end()) {
       cout << "\"" << sub << "\" is part of \"" << s << "\""
```

```
            << endl;
    }
}
```

Разумеется, точно так же можно вызывать функции-члены (сравните с разделом 10.2.2).

```
// fo/lambda5.cpp

#include <functional>
#include <algorithm>
#include <vector>
#include <iostream>
#include <string>
using namespace std;
using namespace std::placeholders;

class Person {
   private:
      string name;
   public:
      Person (const string& n) : name(n) {
   }

   void print () const {
      cout << name << endl;
   }

   void print2 (const string& prefix) const {
      cout << prefix << name << endl;
   }
   ...
};

int main()
{
   vector<Person> coll
        = { Person("Tick"), Person("Trick"), Person("Track") };

   // вызываем функцию-член print() каждого объекта класса Person
   for_each (coll.begin(), coll.end(),
             [] (const Person& p) {
                p.print();
             });
   cout << endl;

   // вызываем функцию-член print2() с дополнительным аргументом
   // из каждого объекта класса Person
   for_each (coll.begin(), coll.end(),
             [] (const Person& p) {
                p.print2("Person: ");
             });
}
```

10.3.4. Лямбда-функции как функции-хеширования, критерий сортировки и критерий эквивалентности

Как указывалось ранее, лямбда-функции можно использовать в качестве хеш-функций, а также критериев порядка и сортировки. Рассмотрим пример:

```
class Person {
   ...
};

auto hash = [] (const Person& p) {
   ...
};



auto eq = [] (const Person& p1, const Person& p2) {
   ...
};

// создаем неупорядоченное множество с пользовательским поведением
unordered_set<Person,decltype(hash),decltype(eq)> pset(10,hash,eq);
```

И снова подчеркнем, что мы должны использовать ключевое слово `decltype` для передачи типа лямбда-функций в контейнер `unordered_set`, потому что он создает их копии. Кроме того, конструктору необходимо передать хеш-функцию и критерий эквивалентности, потому что в противном случае конструктор вызывает конструктор по умолчанию для хеш-функции и критерия эквивалентности, которые для лямбда-функций не определены.

Из-за этих неудобств определение класса для функциональных объектов можно считать более читабельным и даже удобным. Итак, если в программе используется состояние, лямбда-функции не всегда лучше.

Полный пример использования лямбда-функций для определения хеш-функции и критерия эквивалентности для неупорядоченных контейнеров см. в разделе 7.9.7.

Глава 11

Алгоритмы STL

В главе описываются все алгоритмы стандартной библиотеки C++. Она начинается с обзора алгоритмов и общих замечаний, относящихся к ним. Затем излагаются точные описания каждого алгоритма и приводятся один или несколько примеров их использования.

11.1. Заголовочные файлы для алгоритмов

Для использования алгоритмов стандартной библиотеки C++ в программу необходимо включить заголовочный файл `<algorithm>`:

```
#include <algorithm>
```

Помимо определения алгоритмов, этот заголовочный файл содержит несколько вспомогательных функций: `min()`, `max()` и `minmax()`, рассмотренных в разделе 5.5.1, и итераторную функцию `iter_swap()`, описанную в разделе 9.3.4.

Некоторые из алгоритмов STL предназначены для работы с числами. По этой причине они определены в заголовочном файле `<numeric>`:

```
#include <numeric>
```

В целом численные компоненты стандартной библиотеки C++ рассматриваются в главе 17. Однако я решил обсудить численные алгоритмы здесь, потому что, по моему мнению, важнее то, что они являются алгоритмами STL, чем то, что они предназначены для работы с числами.

При использовании алгоритмов часто возникает потребность в функциональных объектах и функциональных адаптерах. Они описаны в главе 10 и определены в заголовочном файле `<functional>`:

```
#include <functional>
```

11.2. Обзор алгоритмов

В разделе приводится обзор всех алгоритмов стандартной библиотеки C++, чтобы дать читателям представление об их возможностях и облегчить поиск лучшего алгоритма для решения поставленной задачи.

11.2.1. Краткое введение

Алгоритмы были впервые рассмотрены в главе 6 вместе с библиотекой STL. В частности, в разделах 6.4 и 6.7 шла речь о роли алгоритмов и было указано несколько важных ограничений, наложенных на их использование. Все алгоритмы STL обрабатывают один или несколько диапазонов итераторов. Первый диапазон обычно задается своим началом и концом. Для дополнительных диапазонов обычно достаточно указать только начало, потому что его конец определяется количеством элементов в первом диапазоне. Вызывающая сторона должна гарантировать, что диапазоны являются корректными. Иначе говоря, начало должно предшествовать концу или совпадать с ним. Дополнительные диапазоны должны содержать достаточное количество элементов.

Алгоритмы работают в режиме замены, а не вставки. Таким образом, вызывающая сторона должна гарантировать, что принимающие диапазоны содержат достаточное количество элементов. Для переключения из режима замены в режим вставки можно использовать специальные итераторы (см. раздел 9.4.2).

Для повышения гибкости и мощи алгоритмов некоторые из них допускают передачу операций, определенных пользователем, которые можно вызывать в алгоритме. Эти операции могут быть обычными функциями или функциональными объектами. Если эти функции возвращают булево значение, то они называются *предикатами*. Предикаты можно использовать для решения следующих задач.

- Можно передать функцию, функциональный объект или лямбда-функцию, определяющую унарный предикат, в качестве критерия поиска в алгоритме поиска. Этот унарный предикат используется для проверки, удовлетворяет ли элемент заданному критерию. Например, можно искать первый элемент, меньший 50.
- Можно передать функцию, функциональный объект или лямбда-функцию, определяющую бинарный предикат, в качестве критерия сортировки в алгоритме сортировки. Этот бинарный предикат используется для сравнения двух элементов. Например, можно передать предикат, позволяющий сортировать по фамилиям объекты, содержащие персональную информацию (см. раздел 10.1.1).
- Можно передать унарный предикат как критерий, указывающий, к каким элементам должна применяться операция. Например, можно указать, что удалять из диапазона следует только элементы с четными значениями.
- Можно задавать операции над числами в численных алгоритмах. Например, можно использовать алгоритм `accumulate()`, который обычно суммирует элементы, для вычисления произведения всех элементов.

Отметим, что предикаты не изменяют свое состояние при вызове функции (см. раздел 10.1.4).

Примеры и подробности, касающиеся функций, функциональных объектов и лямбда-функций, приведены в разделах 6.8–6.10 и главе 10.

11.2.2. Классификация алгоритмов

Разные алгоритмы удовлетворяют разные потребности и могут быть классифицированы по своему предназначению. Например, некоторые алгоритмы только читают элементы, некоторые модифицируют элементы, а некоторые изменяют их порядок. В этом

подразделе кратко описываются функциональные возможности каждого алгоритма и указывается, чем он отличается от похожих алгоритмов.

Первое впечатление о предназначении алгоритма дает его имя. Разработчики библиотеки STL использовали два специальных суффикса.

1. **Суффикс `_if`** используется, когда можно вызвать две формы алгоритма, имеющие одинаковое количество параметров, передавая либо значение, либо функцию, либо функциональный объект. В этом случае для значений используется версия без суффикса, а для функций и функциональных объектов — версия с суффиксом `_if`. Например, алгоритм `find()` ищет элемент с определенным значением, а алгоритм `find_if()` ищет элемент, удовлетворяющий критерию, заданному функцией, функциональным объектом или лямбда-функцией.

   Однако не все алгоритмы, параметром которых является функция или функциональный объект, имеют суффикс `_if`. Если версия алгоритма с функцией или функциональным объектом имеет дополнительный параметр, то она имеет то же самое имя. Например, алгоритм `min_element()`, вызываемый с двумя аргументами, возвращает минимальный элемент в диапазоне в соответствии со сравнением с помощью оператора <. Если передать третий элемент, то он будет использоваться в качестве критерия сравнения.

2. **Суффикс `_copy`** используется для индикации того, что элементы не только обрабатываются, но и копируются в целевой диапазон. Например, алгоритм `reverse()` изменяет порядок следования элементов в диапазоне на противоположный, в то время как алгоритм `reverse_copy()` копирует элементы в другой диапазон в обратном порядке.

В последующих разделах описывают алгоритмы на основе следующей классификации.

- Немодифицирующие алгоритмы.
- Модифицирующие алгоритмы.
- Алгоритмы удаления.
- Перестановочные алгоритмы.
- Алгоритмы сортировки.
- Алгоритмы для упорядоченных диапазонов.
- Численные алгоритмы.

Алгоритмы, относящиеся к нескольким категориям, относятся к категории, которую я считаю наиболее важной.

Немодифицирующие алгоритмы

Немодифицирующие алгоритмы не изменяют ни порядок, ни значения обрабатываемых элементов. Эти алгоритмы работают с итераторами ввода и однонаправленными итераторами; следовательно, их можно применять ко всем стандартным контейнерам. Немодифицирующие алгоритмы стандартной библиотеки C++ приведены в табл. 11.1. Немодифицирующие, предусмотренные для упорядоченных диапазонов, перечислены в подразделе "Алгоритмы сортировки".

Исторически одним из наиболее важных алгоритмов был алгоритм `for_each()`. Он применяет к каждому элементу операцию, заданную вызывающей стороной. Эта операция обычно используется для индивидуальной обработки каждого элемента диапазона. Например, алгоритму `for_each()` можно передать функцию, выводящую на экран каждый элемент или применяющую операцию модификации к каждому элементу. Отметим, впрочем, что начиная со стандарта C++11 более удобным и естественным является цикл по диапазону `for` (см. разделы 3.1.4 и 6.2.1). Таким образом, алгоритм `for_each()` со временем может утратить свое значение.

Некоторые из немодифицирующих алгоритмов выполняют поиск. К сожалению, принципы именования алгоритмов поиска слишком запутаны. Кроме того, принципы именования алгоритмов поиска и функций поиска строк отличаются друг от друга (табл. 11.2). Как это часто бывает, это объясняется историческими причинами. Во-первых, библиотека STL и классы строк разрабатывались независимо друг от друга. Во-вторых, алгоритмы `find_end()`, `find_first_of()` и `search_n()` изначально не были частью библиотеки STL. Например, имя `find_end()` вместо `search_end()` было выбрано случайно (зарывшись в детали, можно легко забыть такие аспекты, как согласованность). Также случайно было выбрано имя для алгоритма `search_n()`, нарушающего общую концепцию исходной библиотеки STL. Описание этой проблемы изложено в разделе 11.5.3.

Таблица 11.1. Немодифицирующие алгоритмы

Имя	Действие
`for_each()`	Выполняет операцию над каждым элементом
`count()`	Возвращает количество элементов
`count_if()`	Возвращает количество элементов, соответствующих определенному критерию
`min_element()`	Возвращает элемент с наименьшим значением
`max_element()`	Возвращает элемент с наибольшим значением
`minmax_element()`	Возвращает элементы с наибольшим и наименьшим значениями (начиная со стандарта C++11)
`find()`	Ищет первый элемент, равный переданному значению
`find_if()`	Ищет первый элемент, удовлетворяющий определенному критерию
`find_if_not()`	Ищет первый элемент, не удовлетворяющий определенному критерию (по стандарту C++11)
`search_n()`	Ищет первые *n* последовательных элементов с определенными свойствами
`search()`	Ищет первое вхождение подстроки
`find_end()`	Ищет последнее вхождение подстроки
`find_first_of()`	Ищет первый из нескольких возможных элементов
`adjacent_find()`	Ищет два смежных элемента, равных между собой (по определенному критерию)
`equal()`	Возвращает результат проверки равенства двух диапазонов
`is_permutation()`	Возвращает результат проверки, содержат ли два неупорядоченных диапазона равные элементы (начиная со стандарта C++11)

Окончание табл. 11.1

Имя	Действие
`mismatch()`	Возвращает первые отличающиеся друг от друга элементы двух последовательностей
`lexicographical_compare()`	Возвращает результат лексикографической проверки, меньше ли один диапазон другого
`is_sorted()`	Возвращает результат проверки, являются ли упорядоченными элементы в диапазоне (начиная со стандарта C++11)
`is_sorted_until()`	Возвращает первый элемент диапазона, нарушающий порядок (начиная со стандарта C++11)
`is_partitioned()`	Возвращает результат проверки, разделен ли диапазон на две группы по заданному критерию (начиная со стандарта C++11)
`partition_point()`	Возвращает разделяющий элемент для диапазона, разделенного на элементы, удовлетворяющие и не удовлетворяющие заданному предикату (начиная со стандарта C++11)
`is_heap()`	Возвращает результат проверки, упорядочены ли элементы в диапазоне согласно свойству пирамиды (начиная со стандарта C++11)
`is_heap_until()`	Возвращает первый элемент диапазона, нарушающий свойство пирамиды (начиная со стандарта C++11)
`all_of()`	Возвращает результат проверки того, что все элементы удовлетворяют определенному критерию (начиная со стандарта C++11)
`any_of()`	Возвращает результаты проверки того, что по крайней мере один элемент соответствует заданному критерию (начиная со стандарта C++11)
`none_of()`	Возвращает результат проверки того, что ни один элемент не соответствует заданному критерию (начиная со стандарта C++11)

Таблица 11.2. Сравнение функций и алгоритмов для поиска строк

Поиск	Строковые функции	Алгоритм STL
Первое вхождение элемента	`find()`	`find()`
Последнее вхождение элемента	`rfind()`	`find()` с обратным итератором
Первое вхождение поддиапазона	`find()`	`search()`
Последнее вхождение поддиапазона	`rfind()`	`find_end()`
Первое вхождение нескольких элементов	`find_first_of()`	`find_first_of()`
Последнее вхождение нескольких элементов	`find_last_of()`	`find_first_of()` с обратным итератором
Первое вхождение *n* последовательных элементов		`search_n()`

Модифицирующие алгоритмы

Модифицирующие алгоритмы изменяют значения элементов. Такие алгоритмы могут модифицировать элементы диапазона непосредственно или копируя их в другой диапазон. Если элементы копируются в диапазон-получатель, исходный диапазон не изменяется. Модифицирующие алгоритмы стандартной библиотеки C++ перечислены в табл. 11.3.

Основными модифицирующими алгоритмами являются `for_each()` (снова) и `transform()`. Оба эти алгоритма могут модифицировать элементы последовательности. Однако они работают по-разному.

- Алгоритм **`for_each()`** получает операцию, модифицирующую его аргумент. Таким образом, этот аргумент должен передаваться по ссылке. Например:

```
void square (int& elem) // передача по ссылке
{
    elem = elem * elem; // прямое присваивание вычисленного значения
}
...
for_each(coll.begin(),coll.end(), // диапазон
         square);                 // операция
```

- Алгоритм **`transform()`** использует операцию, возвращающую модифицированный аргумент. С ее помощью можно присвоить результат исходному элементу. Например:

```
int square (int elem)   // передача по  значению
{
    return elem * elem; // возвращает вычисленное значение
}
...
transform (coll.cbegin(), coll.cend(), // диапазон-источник
           coll.begin(),               // диапазон-получатель
           square);                    // операция
```

Таблица 11.3. Модифицирующие алгоритмы

Имя	Действие
`for_each()`	Применяет операцию к каждому элементу
`copy()`	Копирует диапазон начиная с первого элемента
`copy_if()`	Копирует элементы, удовлетворяющие критерию (начиная со стандарта C++11)
`copy_n()`	Копирует *n* элементов (начиная со стандарта C++11)
`copy_backward()`	Копирует диапазон начиная с последнего элемента
`move()`	Перемещает элементы диапазона начиная с первого элемента (начиная со стандарта C++11)
`move_backward()`	Перемещает элементы диапазона начиная с последнего элемента (начиная со стандарта C++11)
`transform()`	Модифицирует (и копирует) элементы; объединяет элементы двух диапазонов

Окончание табл. 11.3

Имя	Действие
`merge()`	Объединяет два диапазона
`swap_ranges()`	Меняет местами два диапазона
`fill()`	Заменяет каждый элемент заданным значением
`fill_n()`	Заменяет *n* элементов заданным значением
`generate()`	Заменяет каждый элемент результатом операции
`generate_n()`	Заменяет *n* элементов результатом операции
`iota()`	Заменяет каждый элемент последовательностью инкрементируемых значений (начиная со стандарта C++11)
`replace()`	Заменяет элементы, имеющие заданное значение, другим значением
`replace_if()`	Заменяет элементы, соответствующие критерию, другим значением
`replace_copy()`	Заменяет элементы, имеющие заданное значение, копируя весь диапазон
`replace_copy_if()`	Заменяет элементы, соответствующие критерию, копируя весь диапазон

Алгоритм `transform()` работает немного медленнее, потому что он возвращает и присваивает результат, а не модифицирует каждый элемент непосредственно. Однако он является более гибким, потому что его можно использовать для модификации элементов в процессе их копирования в другой диапазон. Другая версия алгоритма `transform()` может обрабатывать и объединять элементы двух исходных диапазонов.

Строго говоря, алгоритм `merge()` можно было не включать в список модифицирующих алгоритмов, потому что он требует, чтобы исходные диапазоны были упорядочены. Его следовало бы отнести к алгоритмам для упорядоченных диапазонов. Однако на практике алгоритм `merge()` также объединяет элементы неупорядоченных диапазонов. Разумеется, результат остается неупорядоченным. Тем не менее, для того чтобы избежать возможных неприятностей, алгоритм `merge()` следует применять только к упорядоченным диапазонам.

Отметим, что элементы ассоциативных и неупорядоченных контейнеров являются константными, чтобы пользователь не мог нарушить их упорядоченность из-за модификации. Следовательно, эти контейнеры нельзя использовать в качестве целевых для модифицирующих алгоритмов.

Кроме указанных модифицирующих алгоритмов, стандартная библиотека C++ содержит алгоритмы для упорядоченных диапазонов.

Алгоритмы удаления

Алгоритмы удаления представляют собой специальную разновидность модифицирующих алгоритмов. Они могут удалять элементы либо в отдельном диапазоне, либо при их копировании в другой диапазон. Как и при работе с модифицирующими алгоритмами, ассоциативные или неупорядоченные контейнеры нельзя использовать как целевые для алгоритмов удаления, потому что элементы этих контейнеров считаются константами. Алгоритмы удаления стандартной библиотеки C++ перечислены в табл. 11.4.

Таблица 11.4. Алгоритмы удаления

Имя	Действие
`remove()`	Удаляет элементы с заданным значением
`remove_if()`	Удаляет элементы, удовлетворяющие заданному критерию
`remove_copy()`	Копирует элементы, не равные заданному значению
`remove_copy_if()`	Копирует элементы, не удовлетворяющие заданному критерию
`unique()`	Удаляет соседние дубликаты (элементы, совпадающие со своим предшественником)
`unique_copy()`	Копирует элементы, удаляя соседние дубликаты

Отметим, что эти алгоритмы удаляют элементы только логически, заменяя их следующими, еще не удаленными элементами. Таким образом, алгоритмы удаления не изменяют количество элементов в рабочем диапазоне. Вместо этого они возвращают позицию нового конца этого диапазона. Вызывающая сторона сама должна выбирать способ использования нового конца диапазона, например, для физического удаления элементов (подробнее см. в разделе 6.7.1).

Перестановочные алгоритмы

Перестановочные алгоритмы изменяют порядок элементов (но не их значения), присваивая и обменивая их значения. Перестановочные алгоритмы стандартной библиотеки C++ приведены в табл. 11.5. Как и при работе с модифицирующими алгоритмами, ассоциативные или неупорядоченные контейнеры нельзя использовать в качестве целевых для алгоритмов удаления, потому что элементы этих контейнеров считаются константами.

Таблица 11.5. Перестановочные алгоритмы

Имя	Действие
`reverse()`	Изменяет порядок следования элементов на противоположный
`reverse_copy()`	Копирует элементы, изменяя порядок их следования на противоположный
`rotate()`	Производит циклический сдвиг элементов
`rotate_copy()`	Копирует элементы, выполняя циклический сдвиг
`next_permutation()`	Выполняет перестановку элементов
`prev_permutation()`	Выполняет перестановку элементов
`shuffle()`	Перетасовывает элементы в случайном порядке (начиная со стандарта C++11)
`random_shuffle()`	Перетасовывает элементы в случайном порядке
`partition()`	Изменяет порядок следования элементов так, что элементы, удовлетворяющие заданному критерию, оказываются впереди
`stable_partition()`	Делает то же, что и `partition()`, сохраняя относительный порядок элементов, удовлетворяющих и не удовлетворяющих заданному критерию
`partition_copy()`	Копирует элементы, изменяя порядок их следования так, что элементы, удовлетворяющие заданному критерию, оказываются впереди

Алгоритмы сортировки

Алгоритмы сортировки представляют собой специальную разновидность перестановочных алгоритмов, поскольку они изменяют порядок следования элементов. Однако алгоритмы сортировки более сложные, а значит, обычно работают дольше, чем обычные перестановочные алгоритмы. Как правило, сложность этих алгоритмов больше линейной[1] и требует использования итераторов произвольного доступа (для целевых диапазонов). Алгоритмы сортировки приведены в табл. 11.6, а в табл. 11.7 — соответствующие алгоритмы, позволяющие проверять, является ли последовательность (частично) упорядоченной.

Таблица 11.6. Алгоритмы сортировки

Имя	Действие
`sort()`	Упорядочивает все элементы
`stable_sort()`	Упорядочивает элементы, сохраняя порядок следования одинаковых элементов
`partial_sort()`	Упорядочивает элементы, пока первые *n* элементов не будут следовать в требуемом порядке
`partial_sort_copy()`	Копирует и упорядочивает элементы
`nth_element()`	Упорядочивает элементы относительно *n*-й позиции
`partition()`	Изменяет порядок следования элементов так, чтобы элементы, удовлетворяющие заданному критерию, оказались впереди
`stable_partition()`	Делает то же, что и `partition()`, сохраняя относительный порядок элементов, удовлетворяющих и не удовлетворяющих заданному критерию
`partition_copy()`	Копирует элементы, изменяя порядок их следования так, что элементы, удовлетворяющие заданному критерию, оказываются впереди
`make_heap()`	Превращает диапазон в пирамиду
`push_heap()`	Добавляет элемент в пирамиду
`pop_heap()`	Удаляет элемент из пирамиды
`sort_heap()`	Упорядочивает пирамиду (после вызова она уже не будет пирамидой)

Таблица 11.7. Алгоритмы проверки сортированности

Имя	Действие
`is_sorted()`	Возвращает результат проверки, что элементы диапазона упорядочены (начиная со стандарта C++11)
`is_sorted_until()`	Возвращает первый неупорядоченный элемент в диапазоне (начиная со стандарта C++11)
`is_partitioned()`	Возвращает результат проверки, что элементы в диапазоне разделены на две группы по заданному критерию (начиная со стандарта C++11)

[1] Понятие сложности алгоритма обсуждается в разделе 2.2.

Окончание табл. 11.7

Имя	Действие
`partition_point()`	Возвращает разделяющий элемент для диапазона, разделенного на элементы, удовлетворяющие предикату, и элементы, не удовлетворяющие предикату (начиная со стандарта C++11)
`is_heap()`	Возвращает результат проверки, являются ли элементы диапазона упорядоченными в соответствии со свойством пирамиды (начиная со стандарта C++11)
`is_heap_until()`	Возвращает первый элемент диапазона, не упорядоченного в соответствии со свойством пирамиды (начиная со стандарта C++11)

Для алгоритмов сортировки время выполнения часто является самой важной характеристикой. По этой причине стандартная библиотека C++ содержит несколько алгоритмов сортировки. Эти алгоритмы используют разные способы сортировки, причем некоторые их них упорядочивают только часть элементов. Например, алгоритм `nth_element()` останавливает работу, когда *n*-й элемент последовательности удовлетворяет заданному критерию сортировки. Для других элементов он гарантирует только то, что значения предшествующих элементов не превышают значение *n*-го элемента, а последующие больше или равны.

Для того чтобы упорядочить все элементы последовательности, следует применять следующие алгоритмы.

- Алгоритм **`sort()`**, исторически основанный на алгоритме быстрой сортировки (*quicksort*). Следует иметь в виду, что текущие реализации используют алгоритм *introsort* — новый алгоритм, который по умолчанию работает как алгоритм *quicksort*, но в случае квадратичной сложности превращается в алгоритм *heapsort*. Таким образом, этот алгоритм гарантирует высокое быстродействие (в среднем $n*\log(n)$).

  ```
  // упорядочивает все элементы
  // - сложность в лучшем случае n*log(n)
  sort (coll.begin(), coll.end());
  ```

  До появления стандарта C++11 алгоритм `sort()` гарантировал лишь сложность $n*\log(n)$ в среднем и квадратичную сложность в худшем случае.

  Для того чтобы избежать худшего случая, следует использовать другой алгоритм, например `partial_sort()` или `stable_sort()`.

- Алгоритм **`partial_sort()`**, исторически основанный на алгоритме пирамидальной сортировки (*heapsort*). Таким образом, в любом случае он гарантирует сложность $n*log(n)$. Однако в большинстве случаев алгоритм *heapsort* работает от двух до пяти раз медленнее, чем алгоритм *quicksort*. Итак, если алгоритм `sort()` реализован как *quicksort*, а `partial_sort()` реализован как *heapsort*, то `partial_sort()` имеет меньшую сложность, а `sort()` — более высокую производительность в большинстве случаев. Преимущество алгоритма `partial_sort()` заключается в том, что он гарантирует сложность $n*log(n)$ в любом случае и никогда не достигает квадратичной сложности.

  Кроме того, алгоритм `partial_sort()` имеет специальную возможность останавливать сортировку, когда требуется упорядочить только первые *n* элементов.

Для того чтобы упорядочить все элементы, необходимо передать в качестве второго и последнего аргумента конец последовательности.

```
// сортируем все элементы
// - сложность всегда n*log(n)
// - но обычно вдвое медленнее, чем алгоритм sort()
partial_sort (coll.begin(), coll.end(), coll.end());
```

- Алгоритм **`stable_sort()`**, исторически основанный на алгоритме *mergesort*. Он сортирует все элементы.

```
// сортируем все элементы
// - сложность n*log(n) или n*log(n)*log(n)
stable_sort (coll.begin(), coll.end());
```

Однако, для того чтобы обеспечить сложность $n*log(n)$, ему требуется дополнительная память. В противном случае его сложность составляет лишь $n*log(n)*log(n)$. Преимущество алгоритма `stable_sort()` заключается в том, что он сохраняет порядок следования одинаковых элементов.

Теперь читатели имеют представление о том, какой алгоритм сортировки лучше всего подходит для решения их задач. Однако это еще не все. Стандарт гарантирует определенную сложность, но не указывает, как ее достичь. Это дает разработчикам преимущество, поскольку можно изобрести новые, более эффективные способы сортировки, не нарушая стандарта. Например, алгоритм `sort()` изначально был реализован с помощью алгоритма *quicksort*. По этой причине в стандарте C++98 была указана сложность "$n*\log(n)$ в среднем", потому что алгоритм *quicksort* в худшем случае становится квадратичным. Однако, даже в реализациях стандартов C++98 и C++03 использовался алгоритм *introsort*, чтобы использовать его преимущества, поэтому в стандарте C++11 ограничение "в среднем" было удалено. Недостаток того, что стандарт не регламентирует точную сложность, состоит в том, что реализация может использовать соответствующий стандарту, но совершенно неудачный алгоритм. Например, использование алгоритма *heapsort* для реализации алгоритма `sort()` вполне соответствует стандарту. Разумеется, можно просто протестировать алгоритмы и выбрать лучший, но средства измерения эффективности могут зависеть от платформы.

Помимо вышеуказанных, существует множество алгоритмов для сортировки элементов. Например, алгоритмы для работы с пирамидой вызывают функции, непосредственно реализующие пирамиду (ее можно рассматривать как бинарное дерево, реализованное как последовательная коллекция). Алгоритмы для работы с пирамидой являются основой для создания эффективных реализаций приоритетных очередей (см. раздел 12.3). Эти алгоритмы можно использовать для сортировки всех элементов коллекции, вызывая их следующим образом:

```
// сортируем все элементы
// - сложность n+n*log(n)
make_heap (coll.begin(), coll.end());
sort_heap (coll.begin(), coll.end());
```

Подробное описание пирамиды и алгоритмов для работы с ней приведено в разделе 11.9.4.

Алгоритмы `nth_element()` используются, когда нужен только n-й упорядоченный элемент или требуется найти множество из n старших или n младших элементов (не упорядоченных). Таким образом, алгоритм `nth_element()` позволяет разделить элементы

на два подмножества в соответствии с критерием сортировки. Однако для решения этой задачи можно использовать также алгоритмы `partition()` и `stable_partition()`. Разница заключается в следующем.

- Алгоритму **`nth_element()`** передается количество элементов, которые должна содержать первая часть (а значит, и вторая). Например:

```
// перемещает четыре младших элемента вперед
nth_element (coll.begin(),       // начало диапазона
             coll.begin()+3,     // позиция между первой и второй частями
             coll.end());        // конец диапазона
```

  Но после вызова точная разница между первой и второй частями неизвестна. На самом деле обе части могут содержать в качестве *n*-го элемента одно и то же значение.

- Алгоритму **`partition()`** передается точный критерий сортировки, который определяет различие между первой и второй частями. Например:

```
// перемещаем вперед все элементы, меньше семи
vector<int>::iterator pos;
pos = partition (coll1.begin(), coll1.end(), // диапазон
                 [](int elem){               // критерий
                 return elem<7;
});
```

  В этом случае после вызова невозможно узнать, сколько именно элементов содержат первая и вторая части. Значение `pos`, возвращаемое алгоритмом, ссылается на первый элемент второй части, содержащей все элементы, не удовлетворяющие критерию, если они есть.

- Алгоритм **`stable_partition()`** работает аналогично алгоритму `partition()`, но имеет дополнительные возможности. Он гарантирует, что порядок элементов в обеих частях остается неизменным по отношению к другим элементам в той же части.

Критерий сортировки всегда можно передать любому алгоритму сортировки в качестве необязательного аргумента. По умолчанию в качестве критерия сортировки используется функциональный объект `less<>`, так что элементы упорядочиваются в возрастающем порядке. Отметим, что критерий сортировки должен определять *строгое слабое упорядочение* значений. Критерий, в котором значения сравниваются с помощью операции "меньше или равно", не соответствует этому требованию (см. раздел 7.7).

Как и при работе с модифицирующими алгоритмами, в качестве целевого контейнера нельзя использовать ассоциативные контейнеры, потому что их элементы считаются константами.

Списки и последовательные списки не имеют итераторов произвольного доступа, поэтому к ним нельзя применять алгоритмы сортировки. Однако для сортировки их элементов предусмотрена функция-член `sort()` (см. раздел 8.8.1).

Алгоритмы для упорядоченных диапазонов

Алгоритмы для упорядоченных диапазонов требуют, чтобы диапазоны, с которыми они работают, были упорядочены в соответствии с заданным критерием сортировки (табл. 11.8). Для ассоциативных контейнеров эти алгоритмы обеспечивают меньшую сложность.

Первые пять алгоритмов для упорядоченных диапазонов в табл. 11.8 являются немодифицирующими, выполняя только поиск по заданному критерию. Другие алгоритмы объединяют два упорядоченных входных диапазона и записывают результат в целевой диапазон. Как правило, результат этих алгоритмом также является упорядоченным.

Таблица 11.8. Алгоритмы сортировки диапазонов

Имя	Действие
`binary_search()`	Возвращает результат проверки, содержит ли диапазон заданный элемент
`includes()`	Возвращает результат проверки, принадлежит ли каждый элемент диапазона другому диапазону
`lower_bound()`	Находит первый элемент, который больше или равен заданному значению
`upper_bound()`	Находит первый элемент, который больше заданного значения
`equal_range()`	Возвращает диапазон элементов, равных заданному значению
`merge()`	Объединяет элементы двух диапазонов
`set_union()`	Создает упорядоченное объединение двух диапазонов
`set_intersection()`	Создает упорядоченное пересечение двух диапазонов
`set_difference()`	Создает упорядоченный диапазон, содержащий все элементы диапазона, не являющиеся элементами другого диапазона
`set_symmetric_difference()`	Создает упорядоченный диапазон, содержащий все элементы, принадлежащие только одному из двух диапазонов
`inplace_merge()`	Объединяет два последовательных упорядоченных диапазона
`partition_point()`	Возвращает элемент, разделяющий диапазон на элементы, удовлетворяющие и не удовлетворяющие предикату (начиная со стандарта C++11)

Численные алгоритмы

Эти алгоритмы объединяют числовые элементы разными способами. Численные алгоритмы стандартной библиотеки C++ приведены в табл. 11.9. Предназначение этих алгоритмов легко распознать по их именам. Они являются более гибкими и мощными, чем может показаться на первый взгляд. Например, по умолчанию алгоритм `accumulate()` вычисляет сумму всех элементов. Но с помощью этого алгоритма можно выполнять конкатенацию строк. Переключаясь с оператора `+` на оператор `*`, можно вычислить произведение всех элементов. В качестве другого примера можно привести алгоритмы `adjacent_difference()` и `partial_sum()`, превращающие диапазон абсолютных значений в диапазон относительных значений, и наоборот.

Таблица 11.9. Численные алгоритмы

Имя	Действие
`accumulate()`	Объединяет значения всех элементов (вычисляет сумму, произведение и т.д.)
`inner_product()`	Объединяет все элементы двух диапазонов
`adjacent_difference()`	Объединяет каждый элемент с его предшественником
`partial_sum()`	Объединяет каждый элемент со всеми его предшественниками

Алгоритмы `accumulate()` и `inner_product()` вычисляют и возвращают одно значение, не изменяя диапазоны. Другие алгоритмы записывают результаты в целевой диапазон, содержащий столько же элементов, сколько содержит исходный диапазон.

11.3. Вспомогательные функции

Оставшаяся часть главы посвящена подробному описанию алгоритмов и содержит по крайней мере один пример использования каждого алгоритма. Для упрощения примеров я использую несколько вспомогательных функций, позволяющих сконцентрировать внимание на сути примера.

```
// algo/algostuff.hpp

#ifndef ALGOSTUFF_HPP
#define ALGOSTUFF_HPP

#include <array>
#include <vector>
#include <deque>
#include <list>
#include <forward_list>
#include <set>
#include <map>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>
#include <iterator>
#include <functional>
#include <numeric>
#include <iostream>
#include <string>

// INSERT_ELEMENTS (collection, first, last)
// - заполняет коллекцию значениями от first до last
// - ЗАМЕЧАНИЕ: диапазон НЕ ЯВЛЯЕТСЯ полуоткрытым
template <typename T>
inline void INSERT_ELEMENTS (T& coll, int first, int last)
{
   for (int i=first; i<=last; ++i) {
        coll.insert(coll.end(),i);
```

```
    }
}

// PRINT_ELEMENTS()
// - выводит на экран необязательную строку optcstr, за которой следуют
// - все элементы коллекции coll,
// - разделенные пробелами

template <typename T>
inline void PRINT_ELEMENTS (const T& coll,
                            const std::string& optcstr="")
{
   std::cout << optcstr;
   for (auto elem : coll) {
        std::cout << elem << ' ';
   }
   std::cout << std::endl;
}

// PRINT_MAPPED_ELEMENTS()
// - выводит на экран необязательную строку optcstr, за которой следуют
// - все элементы коллекции пар "ключ-значение" coll,
// - разделенные пробелами
template <typename T>
inline void PRINT_MAPPED_ELEMENTS (const T& coll,
                                   const std::string& optcstr="")
{
   std::cout << optcstr;
   for (auto elem : coll) {
        std::cout << '[' << elem.first
                  << ',' << elem.second << "] ";
   }
   std::cout << std::endl;
}

#endif /*ALGOSTUFF_HPP*/
```

Во-первых, заголовочный файл `algostuff.hpp` включает все заголовочные файлы, которые могут понадобиться для реализации примеров, так что программе это делать не надо. Во-вторых, он определяет три вспомогательные функции[2].

1. `INSERT_ELEMENTS()` вставляет элементы в контейнер, переданный как первый аргумент. Этим элементам присваиваются значения из диапазона, начало которого задано вторым аргументом, а конец — третьим. Оба значения аргумента включаются, т.е. диапазон *не является* полуоткрытым.
2. `PRINT_ELEMENTS()` выводит на экран все элементы контейнера, переданного как первый аргумент, и разделяет их пробелами. В качестве необязательного второго аргумента можно передать строку, которая используется как префикс перед элементами (см. раздел 6.6).

[2] Начиная со стандарта C++11 функция `PRINT_MAPPED_ELEMENTS()` могла быть определена как частичная специализация функции `PRINT_ELEMENTS()`. Однако, чтобы не использовать слишком много языковых конструкций, обе функции определены по отдельности.

3. `PRINT_MAPPED_ELEMENTS()` делает то же самое для контейнеров, содержащих пары "ключ–значение": отображений, мультиотображений, неупорядоченного отображения и неупорядоченного мультиотображения.

11.4. Алгоритм `for_each()`

Алгоритм `for_each()` очень гибкий, потому что позволяет получить доступ, вычислить и модифицировать каждый элемент разными способами. Однако начиная со стандарта C++11 это удобнее и естественнее сделать с помощью цикла по диапазону `for` (см. разделы 3.1.4 и 6.2.1). Таким образом, алгоритм `for_each()` со временем может утратить свою значимость.

`UnaryProc`
`for_each` (`InputIterator` *beg*, `InputIterator` *end*, `UnaryProc` *op*)

- Выполняет операцию *op* (*elem*) для каждого элемента диапазона [*beg*,*end*).
- Возвращает (внутренне модифицированную) копию аргумента *op*. Начиная со стандарта C++11 возвращаемое значение *op* перемещается.
- Унарная операция *op* может модифицировать элементы. Однако по сравнению с алгоритмом `transform()` делает это несколько иным способом (см. раздел 11.2.2).
- Любое значение, возвращаемое операцией *op*, игнорируется.
- Реализация алгоритма `for_each()` приведена в разделе 6.10.1.
- Сложность: линейная (*numElems* вызовов операции *op*()).

В следующем примере алгоритм `for_each()` передает каждый элемент лямбда-функции, которая выводит его на печать. Таким образом, каждый вызов означает вывод аргумента на печать.

```
// algo/foreach1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll;

   INSERT_ELEMENTS(coll,1,9);

   // вызывает функцию print() для каждого элемента
   for_each (coll.cbegin(), coll.cend(), // диапазон
             [](int elem){               // операция
             cout << elem << ' ';
             });
   cout << endl;
}
```

Эта программа выводит на экран следующий результат:

```
1 2 3 4 5 6 7 8 9
```

Вместо лямбда-функции можно было бы передать обычную функцию, которая вызывалась бы для каждого элемента.

```
void print (int elem)
{
   cout << elem << ' ';
}
...
for_each (coll.cbegin(), coll.cend(), // диапазон
          print);                     // операция
```

Однако следует подчеркнуть, что начиная со стандарта C++11 использование диапазонного цикла `for` является более удобным:

```
for (auto elem : coll) {
     cout << elem << ' ';
}
```

Следующий пример показывает, как модифицировать каждый элемент:

```
// algo/foreach2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll;

   INSERT_ELEMENTS(coll,1,9);

   // добавляем 10 к каждому элементу
   for_each (coll.begin(), coll.end(), // диапазон
             [](int& elem){            // операция
             elem += 10;
             });

   PRINT_ELEMENTS(coll);

   // добавляем значение первого элемента к каждому элементу
   for_each (coll.begin(), coll.end(), // диапазон
             [=](int& elem){           // операция
             elem += *coll.begin();
             });

   PRINT_ELEMENTS(coll);
}
```

Программа выводит на экран следующий результат:

```
11 12 13 14 15 16 17 18 19
22 23 24 25 26 27 28 29 30
```

Как видим, необходимо объявить переменную `elem` как ссылку, чтобы модифицировать ее и определить захват, например `[=]`, так, чтобы появилась возможность добавлять копию первого элемента.

```
for_each (coll.begin(), coll.end(), // диапазон
          [=](int& elem){           // операция
          elem += *coll.begin();
          });
```

Если вместо этого передать ссылку на первый элемент при втором вызове алгоритма `for_each()`

```
for_each (coll.begin(), coll.end(), // диапазон
          [&](int& elem){           // операция
          elem += *coll.begin();
          });
```

то добавляемое значение изменилось бы при вычислении элементов, что привело бы к следующим результатам:

```
11 12 13 14 15 16 17 18 19
22 34 35 36 37 38 39 40 41
```

Кроме того, можно было бы определить обычный функциональный объект

```
// функциональный объект, добавляющий значение, которым он инициализирован
template <typename T>
class AddValue {
   private:
      T theValue; // добавляемое значение
   public:
      // конструктор, инициализирующий добавляемое значение
      AddValue (const T& v) : theValue(v) {
   }

   // вызов функции для добавления значения к элементу
   void operator() (T& elem) const {
      elem += theValue;
   }
}
```

и передать его алгоритму `for_each()`:

```
for_each (coll.begin(), coll.end(), // диапазон
          AddValue<int>(10));       // операция
...
for_each (coll.begin(), coll.end(),      // диапазон
          AddValue<int>(*coll.begin())); // операция
```

Класс `AddValue<>` определяет функциональные объекты, добавляющие значение к каждому элементу, передаваемому в конструктор. Подробное описание этого примера см. в разделе 6.10.1.

Отметим, что то же самое можно было бы сделать с помощью алгоритма `transform()` (см. раздел 11.6.3):

```
// добавляем 10 к каждому элементу
transform (coll.cbegin(), coll.cend(), // диапазон-источник
           coll.begin(),               // диапазон-получатель
           [](int elem){               // операция
              return elem + 10;
           });
...

// добавляем значение первого элемента к каждому элементу
transform (coll.cbegin(), coll.cend(), // диапазон-источник
           coll.begin(),               // диапазон-получатель
           [=](int elem){              // операция
               return elem + *coll.begin();
            });
```

Сравнение алгоритмов `for_each()` и `transform()` см. в разделе 11.2.2.

Третий пример демонстрирует использование значения, возвращаемого алгоритмом `for_each()`. Поскольку алгоритм `for_each()` может возвращать свою операцию, можно вычислить и вернуть результат в этой операции:

```
// algo/foreach3.cpp

#include "algostuff.hpp"
using namespace std;

// функциональный объект для вычисления среднего значения
class MeanValue {
private:
   long num; // количество элементов
   long sum; // сумма всех значений элементов
public:
   // конструктор
   MeanValue () : num(0), sum(0) {
}

   // вызов функции
   // - вычисление одного или нескольких элементов последовательности
   void operator() (int elem) {
      num++;          // увеличивает счетчик на единицу
      sum += elem;    // добавляем значение
   }

   // возвращаем среднее значение (неявное преобразование типа)
   operator double() {
      return static_cast<double>(sum) / static_cast<double>(num);
   }
};

int main()
{
   vector<int> coll;

   INSERT_ELEMENTS(coll,1,8);
```

```
    // вычисляем и выводим на экран среднее значение
    double mv = for_each (coll.begin(), coll.end(), // диапазон
                          MeanValue());             // операция
    cout << "mean value: " << mv << endl;
}
```

Программа выводит на экран следующий результат:

```
mean value: 4.5
```

Здесь можно было бы использовать лямбда-функцию и передать возвращаемое значение по ссылке. Однако в этом сценарии лямбда-функции не обязательно лучше, потому что функциональный объект действительно инкапсулирует и переменную `sum` как внутреннее состояние, и результат заключительного деления этой суммы на количество элементов (см. раздел 10.1.3).

11.5. Немодифицирующие алгоритмы

Алгоритмы, представленные в разделе, открывают доступ к элементам, не модифицируя их значений и не изменяя порядок их следования.

11.5.1. Подсчет элементов

difference_type
count (`InputIterator` *beg*, `InputIterator` *end*, `const T&` *value*)
difference_type
count_if (`InputIterator` *beg*, `InputIterator` *end*, `UnaryPredicate` *op*)

- Первая форма подсчитывает элементы в диапазоне [*beg*,*end*), равные значению *value*.
- Вторая форма подсчитывает элементы в диапазоне [*beg*,*end*), для которых унарный предикат *op*(*elem*) возвращает значение `true`.
- Тип возвращаемого значения *difference_type* представляет собой тип разности для итератора.

  ```
  typename iterator_traits<InputIterator>::difference_type
  ```

- (О свойствах итераторов см. в разделе 9.5.)
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Ассоциативные и неупорядоченные контейнеры имеют аналогичную функцию-член `count()`, предназначенную для подсчета элементов, ключ которых равен определенному значению (см. раздел 8.3.3).
- Сложность: линейная (*numElems* сравнений или вызовов предиката *op*() соответственно).

В следующем примере программа подсчитывает элементы в соответствии с разными критериями:

```
// algo/count1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll;

   int num;
   INSERT_ELEMENTS(coll,1,9);
   PRINT_ELEMENTS(coll,"coll:");

   // подсчитываем элементы, равные 4
   num = count (coll.cbegin(), coll.cend(), // диапазон
                4); // value
   cout << "number of elements equal to 4: " << num << endl;

   // подсчитываем элементы с четными значениями
   num = count_if (coll.cbegin(), coll.cend(), // диапазон
                   [](int elem){ // criterion
                      return elem%2==0;
                   });
   cout << "number of elements with even value: " << num << endl;
   // подсчитываем элементы, значение которых больше 4
   num = count_if (coll.cbegin(), coll.cend(), // range
                   [](int elem){ // criterion
                   return elem>4;
                  });
   cout << "number of elements greater than 4: " << num << endl;
}
```

Программа выводит следующий результат:

```
coll: 1 2 3 4 5 6 7 8 9
number of elements equal to 4: 1
number of elements with even value: 4
number of elements greater than 4: 5
```

Вместо использования лямбда-функции, проверяющей четность элемента, можно было бы использовать привязку

```
   std::bind(std::logical_not<bool>(),
             std::bind(std::modulus<int>(),std::placeholders::_1,2)));
```

или даже устаревшее выражение:

```
std::not1(std::bind2nd(std::modulus<int>(),2))
```

Детали, относящиеся к таким выражениям, см. в разделе 10.2.4.

11.5.2. Минимум и максимум

```
ForwardIterator
min_element (ForwardIterator beg, ForwardIterator end)

ForwardIterator
min_element (ForwardIterator beg, ForwardIterator end, CompFunc op)

ForwardIterator
max_element (ForwardIterator beg, ForwardIterator end)

ForwardIterator
max_element (ForwardIterator beg, ForwardIterator end, CompFunc op)

pair<ForwardIterator,ForwardIterator>
minmax_element (ForwardIterator beg, ForwardIterator end)

pair<ForwardIterator,ForwardIterator>
minmax_element (ForwardIterator beg, ForwardIterator end, CompFunc op)
```

- Эти алгоритмы возвращают позицию минимального или максимального элементов, или пару, состоящую из минимального и максимального элементов в диапазоне [*beg*,*end*).
- Версии без предиката *op* сравнивают элементы с помощью оператора <.
- Для сравнения двух элементов используется предикат *op* (*elem1*, *elem2*).
- Он должен возвращать значение `true`, если первый элемент меньше второго.
- Если в диапазоне существует несколько минимальных или максимальных элементов, алгоритмы `min_element()` и `max_element()` возвращают первый найденный минимальный или максимальный элемент; алгоритм `minmax_element()` возвращает первый минимальный, но последний максимальный элемент, поэтому алгоритмы `max_element()` и `minmax_element()` возвращают разные максимальные элементы.
- Если диапазон пустой, то алгоритмы возвращают позицию *beg* или объект класса `pair<`*beg*, *beg*`>`.
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Сложность: линейная (*numElems*`-1` сравнений или вызовов предиката *op*`()` соответственно для алгоритмов `min_element()` и `max_element()` и 3/2 (*numElems*`-1`) сравнений или вызовов предиката *op*`()` соответственно для алгоритма `minmax_element()`).

Следующая программа выводит минимальные и максимальные элементы в контейнере `coll` с помощью алгоритмов `min_element()` и `max_element()`, а также алгоритма `minmax_element()` и, используя функцию `absLess()`, выводит на печать максимальное и минимальное абсолютные значения.

```
// algo/minmax1.cpp

#include <cstdlib>
#include "algostuff.hpp"
using namespace std;
bool absLess (int elem1, int elem2)
```

```
{
   return abs(elem1) < abs(elem2);
}

int main()
{
   deque<int> coll;

   INSERT_ELEMENTS(coll,2,6);
   INSERT_ELEMENTS(coll,-3,6);

   PRINT_ELEMENTS(coll);

   // обрабатываем контейнер и выводим на печать минимум и максимум
   cout << "minimum: "
        << *min_element(coll.cbegin(),coll.cend())
        << endl;
   cout << "maximum: "
        << *max_element(coll.cbegin(),coll.cend())
        << endl;

   // выводим на печать минимум и максимум, а также расстояние между ними,
   // используя алгоритм minmax_element()
   auto mm = minmax_element(coll.cbegin(),coll.cend());
   cout << "min: " << *(mm.first) << endl;  // Вывод минимума
   cout << "max: " << *(mm.second) << endl; // Вывод максимума
   cout << "distance: " << distance(mm.first,mm.second) << endl;

   // обрабатываем контейнер и выводим на печать
   // минимальное и максимальное абсолютные значения
   cout << "minimum of absolute values: "
        << *min_element(coll.cbegin(),coll.cend(),
                        absLess)
        << endl;
   cout << "maximum of absolute values: "
        << *max_element(coll.cbegin(),coll.cend(),
                        absLess)
        << endl;
}
```

Программа выдает следующие результаты:

```
2 3 4 5 6 -3 -2 -1 0 1 2 3 4 5 6
minimum: -3
maximum: 6
min: -3
max: 6
distance: 9
minimum of absolute values: 0
maximum of absolute values: 6
```

Обратите внимание на то, что эти алгоритмы возвращают *позицию* минимального или максимального элемента соответственно. Следовательно, для вывода значений этих элементом необходимо применить операцию *.

```
auto mm = minmax_element(coll.begin(),coll.end());
cout << "min: " << *(mm.first) << endl;
cout << "max: " << *(mm.second) << endl;
```

Отметим, что алгоритм `minmax_element()` выдает последний максимум, поэтому расстояние (см. раздел 9.3.3) равно 9. Если бы мы использовали алгоритм `max_element()`, то расстояние было бы равно -1.

11.5.3. Поиск элементов

Поиск первого совпадения

```
InputIterator
find (InputIterator beg, InputIterator end, const T& value)
```

```
InputIterator
find_if (InputIterator beg, InputIterator end, UnaryPredicate op)
```

```
InputIterator
find_if_not (InputIterator beg, InputIterator end, UnaryPredicate op)
```

- Первая форма возвращает позицию первого элемента в диапазоне [*beg*,*end*), значение которого равно *value*.
- Вторая форма возвращает позицию первого элемента в диапазоне [*beg*,*end*), для которого унарный предикат *op*(*elem*) возвращает значение `true`.
- Третья форма (начиная со стандарта C++11) возвращает позицию первого элемента в диапазоне [*beg*,*end*), для которого унарный предикат *op*(*elem*) возвращает значение `false`.
- Если искомых элементов нет, то все алгоритмы возвращают позицию *end*.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова. Детали изложены в разделе 10.1.4.
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Если диапазон упорядоченный, то следует использовать алгоритмы `lower_bound()`, `upper_bound()`, `equal_range()` или `binary_search()` (см. раздел 11.10).
- В ассоциативных и неупорядоченных контейнерах есть аналогичная функция-член `find()` (см. раздел 8.3.3), имеющая меньшую сложность (логарифмическую для ассоциативных и даже константную для неупорядоченных контейнеров).
- Сложность: линейная (не более *numElems* сравнений или вызовов предиката *op*() соответственно).

Следующий пример демонстрирует использование алгоритма `find()` для поиска подстроки, начинающейся с первого элемента, равного 4, и заканчивающейся после второго элемента, равного 4, если он есть:

```
// algo/find1.cpp

#include "algostuff.hpp"
using namespace std;
```

```
int main()
{
   list<int> coll;

   INSERT_ELEMENTS(coll,1,9);
   INSERT_ELEMENTS(coll,1,9);
   PRINT_ELEMENTS(coll,"coll: ");

   // найти первый элемент, равный 4
   list<int>::iterator pos1;
   pos1 = find (coll.begin(), coll.end(), // диапазон
               4);                        // значение

   // второй элемент, равный 4
   // - примечание: после первой 4 (если она есть) поиск продолжается
   list<int>::iterator pos2;
   if (pos1 != coll.end()) {
      pos2 = find (++pos1, coll.end(),  // диапазон
                   4);                  // значение
   }

   // выводим все элементы между первой и второй 4 (включая обе)
   // - примечание: теперь нам снова нужна позиция первой 4 (если она есть)
   if (pos1!=coll.end() && pos2!=coll.end()) {
      copy (--pos1, ++pos2,
            ostream_iterator<int>(cout," "));
      cout << endl;
   }
}
```

Для того чтобы найти вторую 4, необходимо прибавить единицу к позиции первой 4. Однако увеличение итератора `end()` на единицу приводит к непредсказуемым последствиям. Следовательно, если вы не уверены, следует выполнять проверку значения, возвращаемого алгоритмом `find()`, прежде чем увеличивать ее на единицу. Программа выводит следующий результат:

```
coll: 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
4 5 6 7 8 9 1 2 3 4
```

Алгоритм `find()` можно вызывать дважды для одного и того же диапазона, но с разными значениями. Однако, используя эти результаты в качестве начала и конца подынтервала, следует проявлять осторожность, поскольку подынтервал может оказаться некорректным. Обсуждение и пример этой проблемы см. в разделе 6.4.1.

Следующий пример демонстрирует использование алгоритмов `find_if()` и `find_if_not()` для поиска элементов по очень разным критериям:

```
// algo/find2.cpp

#include "algostuff.hpp"
using namespace std;
using namespace std::placeholders;
```

```
int main()
{
   vector<int> coll;
   vector<int>::iterator pos;

   INSERT_ELEMENTS(coll,1,9);
   PRINT_ELEMENTS(coll,"coll: ");

   // находим первый элемент, который больше 3
   pos = find_if (coll.begin(), coll.end(),   // диапазон
                  bind(greater<int>(),_1,3)); // критерий

   // выводим его позицию
   cout << "the "
        << distance(coll.begin(),pos) + 1
        << ". element is the first greater than 3" << endl;

   // находим первый элемент, кратный 3
   pos = find_if (coll.begin(), coll.end(),
                  [](int elem){
                     return elem%3==0;
                  });

   // выводим его позицию
   cout << "the "
        << distance(coll.begin(),pos) + 1
        << ". element is the first divisible by 3" << endl;

   // находим первый элемент, который не меньше 5
   pos = find_if_not (coll.begin(), coll.end(),
                      bind(less<int>(),_1,5));
   cout << "first value >=5: " << *pos << endl;
}
```

Первый вызов алгоритма `find_if()` использует простой функциональный объект и адаптер `bind` (см. раздел 10.2.2) для поиска первого элемента, который больше 3. Второй вызов использует лямбда-функцию для поиска первого элемента, кратного 3.

Программа выводит следующий результат:

```
coll: 1 2 3 4 5 6 7 8 9
the 4. element is the first greater than 3
the 3. element is the first divisible by 3
first value >=5: 5
```

Пример поиска первого простого числа с помощью алгоритма `find_if()` приведен в разделе 6.8.2.

Поиск первых *n* последовательных совпадений

```
ForwardIterator
search_n (ForwardIterator beg, ForwardIterator end,
          Size count, const T& value)
```

```
ForwardIterator
search_n (ForwardIterator beg, ForwardIterator end,
          Size count, const T& value, BinaryPredicate op)
```

- Первая форма возвращает позицию первого элемента из *count* элементов в диапазоне [*beg*,*end*), значение которых равно *value*.
- Вторая форма возвращает позицию первого элемента из *count* элементов в диапазоне [*beg*,*end*), для которых бинарный предикат *op* (*elem*, *value*) возвращает значение `true` (значение *value* передается как четвертый аргумент)
- Если искомых элементов нет, то все алгоритмы возвращают позицию *end*.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Эти алгоритмы не являются частью исходной библиотеки STL и не подвергались тщательной проверке. То, что вторая форма использует бинарный, а не унарный предикат, противоречит основным принципам библиотеки STL.
- Сложность: линейная (не более *numElems***count* сравнений или вызовов предиката *op*() соответственно).

Следующий пример демонстрирует поиск последовательных элементов, значение которых равно 7 или является нечетным:

```
// algo/searchn1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   deque<int> coll;

   coll = { 1, 2, 7, 7, 6, 3, 9, 5, 7, 7, 7, 3, 6 };
   PRINT_ELEMENTS(coll);

   // находим три последовательных элемента, равных 7
   deque<int>::iterator pos;
   pos = search_n (coll.begin(), coll.end(), // диапазон
                   3,                        // количество
                   7);                       // значение

   // выводим результат
   if (pos != coll.end()) {
       cout << "three consecutive elements with value 7 "
            << "start with " << distance(coll.begin(),pos) +1
            << ". element" << endl;
   }
   else {
       cout << "no four consecutive elements with value 7 found"
            << endl;
   }
```

```
    // находим четыре последовательных нечетных элемента
    pos = search_n (coll.begin(), coll.end(), // диапазон
                    4,                        // количество
                    0,                        // значение
                    [](int elem, int value){  // критерий
                       return elem%2==1;
                    });

    // выводим результат
    if (pos != coll.end()) {
        cout << "first four consecutive odd elements are: ";
        for (int i=0; i<4; ++i, ++pos) {
             cout << *pos << ' ';
        }
    }
    else {
        cout << "no four consecutive elements with value > 3 found";
    }
        cout << endl;
}
```

Программа выводит следующий результат:

```
1 2 7 7 6 3 9 5 7 7 7 3 6
three consecutive elements with value 7 start with 9. element
first four consecutive odd elements are: 3 9 5 7
```

Вторая форма алгоритма `search_n()` порождает ужасную проблему. Рассмотрим второй вызов алгоритма `search_n()`:

```
    pos = search_n (coll.begin(), coll.end(), // диапазон
                    4,                        // количество
                    0,                        // значение
                    [](int elem, int value){  // критерий
                       return elem%2==1;
                    });
```

Такой вид поиска по специальному критерию противоречит остальной части библиотеки STL. Следуя обычным принципам библиотеки STL, мы должны переписать этот вызов в следующем виде:

```
pos = search_n_if (coll.begin(), coll.end(), // диапазон
                   4,                        // количество
                    [](int elem){            // критерий
                       return elem%2==1;
                    });
```

Однако этот алгоритм требует использования унарного предиката, получающего в качестве второго параметра значение, переданное как четвертый аргумент алгоритма `search_n()`.

К сожалению, никто не заметил этого несоответствия, когда новые алгоритмы включались в стандарт C++98 (они не были частью исходной библиотеки STL). Во-первых, версия с четырьмя аргументами кажется более удобной, потому что ее можно реализовать следующим образом:

```
// находим четыре последовательных элемента, значение которых больше 3
pos = search_n (coll.begin(), coll.end(), // диапазон
                4,                        // количество
                3,                        // значение
                greater<int>());          // критерий
```

Однако, как показывает наш пример, он требует бинарного предиката даже в тех ситуациях, когда можно обойтись унарным.

Как следствие, если вам нужен унарный предикат, например,

```
bool isPrime (int elem);
```

вы должны либо изменить сигнатуру вашей функции, либо написать простую оболочку.

```
bool binaryIsPrime (int elem1, int) {
   return isPrime(elem1);
}
...
pos = search_n (coll.begin(), coll.end(), // диапазон
                4,                        // количество
                0,                        // фиктивное значение
                binaryIsPrime);           // бинарный критерий
```

Поиск первого подынтервала

```
ForwardIterator1
search (ForwardIterator1 beg, ForwardIterator1 end,
        ForwardIterator2 searchBeg, ForwardIterator2 searchEnd)

ForwardIterator1
search (ForwardIterator1 beg, ForwardIterator1 end,
        ForwardIterator2 searchBeg, ForwardIterator2 searchEnd,
        BinaryPredicate op)
```

- Обе формы возвращают позицию первого элемента в первом подынтервале диапазона [*beg*,*end*), соответствующего диапазону [*searchBeg*,*searchEnd*)
- В первой форме элементы подынтервала должны быть равны элементам всего диапазона.
- Во второй форме при каждом сравнении элементов вызывается бинарный предикат *op* (*elem*, *searchElem*), который должен возвращать значение `true`.
- Если искомых элементов нет, то оба алгоритма возвращают позицию *end*.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Поиск подстроки по ее первому и последнему элементам обсуждался в разделе 6.4.1
- Сложность: линейная (не более *numElems*numSearchElems* сравнений или вызовов предиката *op*() соответственно).

Следующий пример демонстрирует поиск последовательности как первой подстроки другой последовательности (сравните с примером использования алгоритма `find_end()`):

```
// algo/search1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   deque<int> coll;
   list<int> subcoll;

   INSERT_ELEMENTS(coll,1,7);
   INSERT_ELEMENTS(coll,1,7);
   INSERT_ELEMENTS(subcoll,3,6);

   PRINT_ELEMENTS(coll, "coll: ");
   PRINT_ELEMENTS(subcoll,"subcoll: ");

   // ищем первое вхождение списка subcoll в дек coll
   deque<int>::iterator pos;
   pos = search (coll.begin(), coll.end(),        // диапазон
                 subcoll.begin(), subcoll.end()); // подынтервал

   // цикл, пока не будет найден список subcoll как подынтервал дека coll
   while (pos != coll.end()) {
      // выводим позицию первого элемента
      cout << "subcoll found starting with element "
           << distance(coll.begin(),pos) + 1
           << endl;

      // ищем следующее вхождение subcoll
      ++pos;
      pos = search (pos, coll.end(),                  // диапазон
                    subcoll.begin(), subcoll.end()); // подынтервал
   }
}
```

Программа выводит следующий результат:

```
coll: 1 2 3 4 5 6 7 1 2 3 4 5 6 7
subcoll: 3 4 5 6
subcoll found starting with element 3
subcoll found starting with element 10
```

Следующий пример демонстрирует использование второй формы алгоритма `search()` для поиска подпоследовательности, соответствующей более сложному критерию. Здесь выполняется поиск подпоследовательности *четное, нечетное и снова четное значение*:

```
// algo/search2.cpp

#include "algostuff.hpp"
using namespace std;

// проверяем четность или нечетность элемента
bool checkEven (int elem, bool even)
```

```
{
   if (even) {
      return elem % 2 == 0;
   }
   else {
      return elem % 2 == 1;
   }
}

int main()
{
   vector<int> coll;

   INSERT_ELEMENTS(coll,1,9);
   PRINT_ELEMENTS(coll,"coll: ");

   // аргументы функции checkEven()
   // - проверка: "четное нечетное четное"
   bool checkEvenArgs[3] = { true, false, true };

   // поиск первого подынтервала в векторе coll
   vector<int>::iterator pos;
   pos = search (coll.begin(), coll.end(),       // диапазон
                 checkEvenArgs, checkEvenArgs+3, // значение подынтервала
                 checkEven);                     // критерий подынтервала

   // цикл, пока не будет найден подынтервал
   while (pos != coll.end()) {
      // выводим позицию первого элемента
      cout << "subrange found starting with element "
           << distance(coll.begin(),pos) + 1
           << endl;

      // поиск следующего подынтервала в векторе coll
      pos = search (++pos, coll.end(),             // диапазон
                    checkEvenArgs, checkEvenArgs+3,  // значения подынтервала
                    checkEven);                      // критерий подынтервала
   }
}
```

Программа выводит следующий результат:

```
coll: 1 2 3 4 5 6 7 8 9
subrange found starting with element 2
subrange found starting with element 4
subrange found starting with element 6
```

Поиск последнего подынтервала

```
ForwardIterator1
find_end (ForwardIterator1 beg, ForwardIterator1 end,
          ForwardIterator2 searchBeg, ForwardIterator2 searchEnd)
```

```
ForwardIterator1
find_end (ForwardIterator1 beg, ForwardIterator1 end,
          ForwardIterator2 searchBeg, ForwardIterator2 searchEnd,
          BinaryPredicate op)
```

- Обе формы возвращают позицию первого элемента в последнем подынтервале диапазона [*beg*,*end*), соответствующем диапазону [*searchBeg*,*searchEnd*)
- В первой форме элементы подынтервала должны быть равны элементам всего диапазона.
- Во второй форме при каждом сравнении элементов вызывается бинарный предикат *op* (*elem*, *searchElem*), который должен возвращать значение `true`.
- Если искомых элементов нет, то оба алгоритма возвращают позицию *end*.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Поиск подстроки по ее первому и последнему элементам обсуждался в разделе 6.4.1
- Эти алгоритмы не были частью исходного библиотеки STL. К сожалению, они вызывают алгоритм `find_end()`, а не `search_end()`, который был бы более уместен, потому что для поиска первого подынтервала используется алгоритм `search()`.
- Сложность: линейная (не более *numElems*numSearchElems* сравнений или вызовов предиката *op*() соответственно).

Следующий пример демонстрирует поиск последовательности как последнего подынтервала другой последовательности (сравните с примером использования алгоритма `search()`):

```
// algo/findend1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   deque<int> coll;
   list<int> subcoll;

   INSERT_ELEMENTS(coll,1,7);
   INSERT_ELEMENTS(coll,1,7);
   INSERT_ELEMENTS(subcoll,3,6);

   PRINT_ELEMENTS(coll, "coll: ");
   PRINT_ELEMENTS(subcoll,"subcoll: ");

   // ищем последнее вхождение списка subcoll в дек coll
   deque<int>::iterator pos;
   pos = find_end (coll.begin(), coll.end(), // диапазон
         subcoll.begin(), subcoll.end());    // подынтервал
```

```
    // цикл, пока subcoll не будет найден как подынтервал coll
    deque<int>::iterator end(coll.end());
    while (pos != end) {
        // выводим позицию первого элемента
        cout << "subcoll found starting with element "
             << distance(coll.begin(),pos) + 1
             << endl;

        // ищем следующее вхождение подынтервала subcoll
        end = pos;
        pos = find_end (coll.begin(), end,                  // диапазон
                        subcoll.begin(), subcoll.end());    // подынтервал
    }
}
```

Программа выводит следующий результат:

```
coll: 1 2 3 4 5 6 7 1 2 3 4 5 6 7
subcoll: 3 4 5 6
subcoll found starting with element 10
subcoll found starting with element 3
```

Вторая форма этого алгоритма иллюстрируется вторым примером использования алгоритма `search()`. Аналогичным образом можно использовать алгоритм `find_end()`.

Поиск первого из нескольких возможных элементов

```
InputIterator
find_first_of (InputIterator beg, InputIterator end,
               ForwardIterator searchBeg, ForwardIterator searchEnd)

InputIterator
find_first_of (InputIterator beg, InputIterator end,
               ForwardIterator searchBeg, ForwardIterator searchEnd,
               BinaryPredicate op)
```

- Первая форма возвращает позицию первого элемента в диапазоне [*beg*,*end*), который также есть в диапазоне [*searchBeg*,*searchEnd*).
- Первая форма возвращает позицию первого элемента в диапазоне [*beg*,*end*), для которого любой вызов

  op(*elem*,*searchElem*)

- для всех элементов диапазона [*searchBeg*,*searchEnd*) выдает `true`.
- Если искомых элементов нет, то оба алгоритма возвращают позицию *end*.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Используя обратные итераторы, можно найти последнее из нескольких возможных значений.
- Эти алгоритмы не являлись частью исходной библиотеки STL.

- До появления стандарта C++11 эти алгоритмы требовали использования однонаправленных итераторов для диапазона [*beg*,*end*), а не итераторов ввода.
- Сложность: линейная (не более *numElems*numSearchElems* сравнений или вызовов предиката *op*() соответственно).

Использование алгоритма `find_first_of()` демонстрируется следующим примером:

```
// algo/findof1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll;
   list<int> searchcoll;

   INSERT_ELEMENTS(coll,1,11);
   INSERT_ELEMENTS(searchcoll,3,5);

   PRINT_ELEMENTS(coll, "coll: ");
   PRINT_ELEMENTS(searchcoll,"searchcoll: ");

   // ищем первое вхождение элемента списка searchcoll в вектор coll
   vector<int>::iterator pos;
   pos = find_first_of (coll.begin(), coll.end(), // диапазон
                        searchcoll.begin(),       // начало поиска
                        searchcoll.end());        // конец поиска
   cout << "first element of searchcoll in coll is element "
        << distance(coll.begin(),pos) + 1
        << endl;

   // ищем первое вхождение элемента списка searchcoll в вектор coll
   vector<int>::reverse_iterator rpos;
   rpos = find_first_of (coll.rbegin(), coll.rend(), // диапазон
                         searchcoll.begin(),         // начало поиска
                         searchcoll.end());          // конец поиска
   cout << "last element of searchcoll in coll is element "
        << distance(coll.begin(),rpos.base())
        << endl;
}
```

Во втором вызове для поиска последнего элемента, значение которого равно значению одного из элементов списка `searchcoll`, используются обратные итераторы. Для вывода позиции элемента вызывается функция-член `base()`, превращающая обратный итератор в обычный. Таким образом, можно вычислить расстояние от начала. Обычно к результату алгоритма `distance()` необходимо добавлять единицу, потому что первый элемент находится на расстоянии `0` от начала, хотя на самом деле является элементом `1`. Однако функция-член `base()` перемещает позицию значения, на которую она ссылается, что приводит к тому же результату (описание функции-члена `base()` см. в разделе 9.4.1).

Программа выводит следующий результат:

```
coll: 1 2 3 4 5 6 7 8 9 10 11
searchcoll: 3 4 5
first element of searchcoll in coll is element 3
last element of searchcoll in coll is element 5
```

Поиск двух смежных элементов с одинаковыми значениями

```
ForwardIterator
adjacent_find (ForwardIterator beg, ForwardIterator end)

ForwardIterator
adjacent_find (ForwardIterator beg, ForwardIterator end,
               BinaryPredicate op)
```

- Первая форма возвращает первый элемент в диапазоне [*beg*,*end*), значение которого равно значению следующего элемента.
- Вторая форма возвращает первый элемент в диапазоне [*beg*,*end*), для которого бинарный предикат *op* (*elem*, *nextElem*) возвращает значение `true`.
- Если искомых элементов нет, то оба алгоритма возвращают позицию *end*.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Сложность: линейная (не более *numElems* сравнений или вызовов предиката *op*()).

Следующая программа демонстрирует использование алгоритма `adjacent_find()`:

```
// algo/adjacentfind1.cpp

#include "algostuff.hpp"
using namespace std;

// проверяем, является ли второй объект вдвое больше первого
bool doubled (int elem1, int elem2)
{
   return elem1 * 2 == elem2;
}

int main()
{
   vector<int> coll;

   coll.push_back(1);
   coll.push_back(3);
   coll.push_back(2);
   coll.push_back(4);
   coll.push_back(5);
   coll.push_back(5);
   coll.push_back(0);

   PRINT_ELEMENTS(coll,"coll: ");
```

```
    // ищем первые два элемента с одинаковыми значениями
    vector<int>::iterator pos;
    pos = adjacent_find (coll.begin(), coll.end());


    if (pos != coll.end()) {
        cout << "first two elements with equal value have position "
             << distance(coll.begin(),pos) + 1
             << endl;
    }

    // ищем первые два элемента, таких, что второй элемент
    // в два раза больше первого
    pos = adjacent_find (coll.begin(), coll.end(), // диапазон
                         doubled);                 // критерий

    if (pos != coll.end()) {
        cout << "first two elements with second value twice the "
             << "first have pos. "
             << distance(coll.begin(),pos) + 1
             << endl;
    }
}
```

Первый вызов алгоритма `adjacent_find()` ищет одинаковые значения. Вторая форма использует алгоритм `doubled()` для поиска первого элемента, такого, что значение следующего элемента в два раза больше. Программа выводит следующий результат:

```
coll: 1 3 2 4 5 5 0
first two elements with equal value have position 5
first two elements with second value twice the first have pos. 3
```

11.5.4. Сравнение диапазонов

Проверка равенства

```
bool
equal (InputIterator1 beg, InputIterator1 end,
       InputIterator2 cmpBeg)

bool
equal (InputIterator1 beg, InputIterator1 end,
       InputIterator2 cmpBeg,
       BinaryPredicate op)
```

- Первая форма проверяет, равны ли элементы диапазона [*beg*,*end*) элементам диапазона, начинающегося с позиции *cmpBeg*.
- Вторая форма проверяет, возвращает ли значение `true` каждый вызов бинарного предиката

op(*elem*, *cmpElem*)

- для соответствующих элементов в диапазоне [*beg*,*end*) и в диапазоне, начинающемся с позиции *cmpBeg*.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Вызывающая сторона должна гарантировать, что диапазон, начинающийся с позиции *cmpBeg*, содержит достаточное количество элементов.
- Для выявления различий в случае несовпадения следует использовать алгоритм `mismatch()`.
- Для того чтобы определить, содержат ли две последовательности одни и те же элементы, только расставленные в разном порядке, стандартом C++11 предусмотрен алгоритм `is_permutation()`.
- Сложность: линейная (не более *numElems* сравнений или вызовов предиката *op*() соответственно).

Следующий пример иллюстрирует обе формы алгоритма `equal()`. Первый вызов проверяет, имеют ли элементы одинаковые значения. Второй вызов использует вспомогательную предикатную функцию для проверки четности и нечетности соответствующих элементов двух коллекций.

```
// algo/equal1.cpp

#include "algostuff.hpp"
using namespace std;

bool bothEvenOrOdd (int elem1, int elem2)
{
   return elem1 % 2 == elem2 % 2;
}

int main()
{
   vector<int> coll1;
   list<int> coll2;

   INSERT_ELEMENTS(coll1,1,7);
   INSERT_ELEMENTS(coll2,3,9);

   PRINT_ELEMENTS(coll1,"coll1: ");
   PRINT_ELEMENTS(coll2,"coll2: ");

   // проверяем, одинаковые ли коллекции
   if (equal (coll1.begin(), coll1.end(), // первый диапазон
              coll2.begin())) {           // второй диапазон
       cout << "coll1 == coll2" << endl;
   }
   else {
       cout << "coll1 != coll2" << endl;
```

```
    }

    // проверяет четность и нечетность соответствующих элементов
    if (equal (coll1.begin(), coll1.end(), // первый диапазон
               coll2.begin(),              // второй диапазон
               bothEvenOrOdd)) {           // критерий сравнения
        cout << "even and odd elements correspond" << endl;
    }
    else {
        cout << "even and odd elements do not correspond" << endl;
    }
}
```

Программы выводит следующий результат:

```
coll1: 1 2 3 4 5 6 7
coll2: 3 4 5 6 7 8 9
coll1 != coll2
even and odd elements correspond
```

Проверка неупорядоченного равенства

```
bool
is_permutation (ForwardIterator1 beg1, ForwardIterator1 end1,
                ForwardIterator2 beg2)
bool
is_permutation (ForwardIterator1 beg1, ForwardIterator1 end1,
                ForwardIterator2 beg2,
                CompFunc op)
```

- Обе формы проверяют, являются ли элемент диапазона [*beg1*,*end1*) перестановкой элементов диапазона, начинающегося с позиции *beg2*; иначе говоря, они проверяют, совпадают ли диапазоны с точностью до перестановки элементов.
- Первая форма сравнивает элементы с помощью оператора ==.
- Вторая форма сравнивает элементы с помощью бинарного предиката *op* (*elem1*, *elem2*), который должен возвращать значение `true`, если элемент *elem1* равен элементу *elem2*.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Все итераторы должны иметь один и тот же тип значения.
- Эти алгоритмы предусмотрены стандартом C++11.
- Сложность: в худшем случае квадратичная (*numElems1* сравнений или вызовов предиката *op* (), если все элементы равны и следуют в одном и том же порядке).

Следующий пример демонстрирует использование неупорядоченного сравнения:

```
// algo/ispermutation1.cpp

#include "algostuff.hpp"
using namespace std;

bool bothEvenOrOdd (int elem1, int elem2)
{
   return elem1 % 2 == elem2 % 2;
}

int main()
{
   vector<int> coll1;
   list<int> coll2;
   deque<int> coll3;

   coll1 = { 1, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
   coll2 = { 1, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
   coll3 = { 11, 12, 13, 19, 18, 17, 16, 15, 14, 11 };

   PRINT_ELEMENTS(coll1,"coll1: ");
   PRINT_ELEMENTS(coll2,"coll2: ");
   PRINT_ELEMENTS(coll3,"coll3: ");

   // проверяем, совпадают ли коллекции с точностью до перестановки
   if (is_permutation (coll1.cbegin(), coll1.cend(), // первый диапазон
                       coll2.cbegin())) {            // второй диапазон
      cout << "coll1 and coll2 have equal elements" << endl;
   }
   else {
      cout << "coll1 and coll2 don't have equal elements" << endl;
   }

   // проверяем количество четных и нечетных элементов
   if (is_permutation (coll1.cbegin(), coll1.cend(), // первый диапазон
                       coll3.cbegin(),               // второй диапазон
                       bothEvenOrOdd)) {             // критерий сравнения
      cout << "numbers of even and odd elements match" << endl;
   }
   else {
      cout << "numbers of even and odd elements don't match" << endl;
   }
}
```

Программа выводит следующий результат:

```
coll1: 1 1 2 3 4 5 6 7 8 9
coll2: 1 9 8 7 6 5 4 3 2 1
coll3: 11 12 13 19 18 17 16 15 14 11
coll1 and coll2 have equal elements
numbers of even and odd elements match
```

Поиск первого различия

```
pair<InputIterator1,InputIterator2>
mismatch (InputIterator1 beg, InputIterator1 end,
          InputIterator2 cmpBeg)
pair<InputIterator1,InputIterator2>
mismatch (InputIterator1 beg, InputIterator1 end,
          InputIterator2 cmpBeg,
          BinaryPredicate op)
```

- Первая форма возвращает первые два различающихся элемента в диапазоне [*beg*,*end*) и диапазоне, начинающемся с позиции *cmpBeg*.
- Вторая форма возвращает первые два различающихся элемента в диапазоне [*beg*,*end*) и диапазоне, начинающемся с позиции *cmpBeg*, для которых предикат *op* (*elem*, *cmpElem*) возвращает значение `false`.
- Если различие не найдено, возвращается объект класса `pair<>`, состоящий из позиции *end* и соответствующего элемента второго диапазона. Отметим, что это не значит, что обе последовательности равны, поскольку вторая последовательность может содержать больше элементов.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Вызывающая сторона должна гарантировать, что диапазон, начинающийся с позиции *cmpBeg*, содержит достаточное количество элементов.
- Для проверки равенства двух диапазонов следует использовать алгоритм `equal()` (см. раздел 11.5.4).
- Сложность: линейная (не более *numElems* сравнений или вызовов предиката *op*() соответственно).

Следующий пример демонстрирует обе формы алгоритма `mismatch()`:

```
// algo/mismatch1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll1 = { 1, 2, 3, 4, 5, 6 };
   list<int> coll2 = { 1, 2, 4, 8, 16, 3 };

   PRINT_ELEMENTS(coll1,"coll1: ");
   PRINT_ELEMENTS(coll2,"coll2: ");

   // ищем первое несовпадение
   auto values = mismatch (coll1.cbegin(), coll1.cend(), // первый диапазон
                           coll2.cbegin());              // второй диапазон
   if (values.first == coll1.end()) {
       cout << "no mismatch" << endl;
```

```
    }
    else {
        cout << "first mismatch: "
             << *values.first << " and "
             << *values.second << endl;
    }

    // ищем первую позицию, в которой элемент коллекции coll1
    // больше соответствующего элемента коллекции coll2
    values = mismatch (coll1.cbegin(), coll1.cend(), // первый диапазон
                       coll2.cbegin(),               // второй диапазон
                       less_equal<int>());           // критерий
    if (values.first == coll1.end()) {
        cout << "always less-or-equal" << endl;
    }
    else {
        cout << "not less-or-equal: "
             << *values.first << " and "
             << *values.second << endl;
    }
}
```

Первый вызов алгоритма `mismatch()` ищет первые несовпадающие элементы. Тип возвращаемого значения:

```
pair<vector<int>::const_iterator,list<int>::const_iterator>
```

Проверяя, равен ли первый элемент в паре итератору, установленному на конец переданного диапазона, мы проверяем, существует ли несовпадение. В данном случае значения соответствующих элементов записываются в стандартный поток вывода.

Второй вызов ищет первую пару элементов, в которой элемент первой коллекции больше соответствующего элемента второй коллекции, и возвращает эти элементы. Программа выводит следующий результат:

```
coll1: 1 2 3 4 5 6
coll2: 1 2 4 8 16 3
first mismatch: 3 and 4
not less-or-equal: 6 and 3
```

Проверка "меньше чем"

```
bool
lexicographical_compare (InputIterator1 beg1, InputIterator1 end1,
                         InputIterator2 beg2, InputIterator2 end2)

bool
lexicographical_compare (InputIterator1 beg1, InputIterator1 end1,
                         InputIterator2 beg2, InputIterator2 end2,
                         CompFunc op)
```

- Обе формы проверяют, что элементы диапазона [*beg1*,*end1*) лексикографически меньше элементов диапазона [*beg2*,*end2*).

- Первая форма сравнивает элементы с помощью оператора <.
- Вторая форма сравнивает элементы с помощью бинарного предиката *op* (*elem1*, *elem2*), который должен возвращать значение `true`, если элемент *elem1* меньше элемента *elem2*.
- *Лексикографическое сравнение* означает, что последовательности сравниваются поэлементно, пока не произойдет одно из следующих событий:
  - если два элемента не равны, результат их сравнения является результатом общего сравнения;
  - если одна последовательность больше не содержит элементов, то исчерпанная последовательность считается меньше другой. Таким образом, сравнение возвращает значение `true`, если первая последовательность больше не содержит элементов;
  - если обе последовательности больше не содержат элементов, то они считаются равными, а результат сравнения равен `false`.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Сложность: линейная (не более `min` (*numElems1*, *numElems2*) сравнений или вызовов предиката *op*() соответственно).

Следующий пример демонстрирует использование лексикографического упорядочения коллекций:

```
// algo/lexicocmp1.cpp

#include "algostuff.hpp"
using namespace std;

void printCollection (const list<int>& l)
{
   PRINT_ELEMENTS(l);
}

bool lessForCollection (const list<int>& l1, const list<int>& l2)
{
   return lexicographical_compare
            (l1.cbegin(), l1.cend(),  // первый диапазон
             l2.cbegin(), l2.cend()); // второй диапазон
}

int main()
{
   list<int> c1, c2, c3, c4;
   // заполняем одинаковыми начальными значениями

   INSERT_ELEMENTS(c1,1,5);
   c4 = c3 = c2 = c1;
```

```
    // а затем разными элементами
    c1.push_back(7);
    c3.push_back(2);
    c3.push_back(0);
    c4.push_back(2);

    // создаем коллекцию коллекций
    vector<list<int>> cc;
    cc.insert ( cc.begin(), { c1, c2, c3, c4, c3, c1, c4, c2 } );

    // выводим все коллекции
    for_each (cc.cbegin(), cc.cend(),
              printCollection);
    cout << endl;

    // упорядочиваем коллекцию лексикографически
    sort (cc.begin(), cc.end(),  // диапазон
          lessForCollection);    // критерий сортировки

    // снова выводим все коллекции
     for_each (cc.cbegin(), cc.cend(),
               printCollection);
}
```

Вектор `cc` инициализируется несколькими коллекциями (все списки). Для сравнения двух коллекций вызов алгоритма `sort()` использует бинарный предикат `lessForCollection()` (описание алгоритма `sort()` приведено в разделе 11.9.1). В функции `lessForCollection()` для лексикографического сравнения коллекций используется алгоритм `lexicographical_compare()`.

Программа выводит следующий результат:

```
1 2 3 4 5 7
1 2 3 4 5
1 2 3 4 5 2 0
1 2 3 4 5 2
1 2 3 4 5 2 0
1 2 3 4 5 7
1 2 3 4 5 2
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5 2
1 2 3 4 5 2
1 2 3 4 5 2 0
1 2 3 4 5 2 0
1 2 3 4 5 7
1 2 3 4 5 7
```

11.5.5. Предикаты для диапазонов

В стандарте C++11 предусмотрены алгоритмы для проверки специальных условий, касающихся конкретного диапазона.

Проверка (частичного) упорядочения

```
bool
is_sorted (ForwardIterator beg, ForwardIterator end)

bool
is_sorted (ForwardIterator beg, ForwardIterator end, BinaryPredicate op)

ForwardIterator
is_sorted_until (ForwardIterator beg, ForwardIterator end)

ForwardIterator
is_sorted_until (ForwardIterator beg, ForwardIterator end,
BinaryPredicate op)
```

- Алгоритм `is_sorted()` проверяет, упорядочены ли элементы в диапазоне [*beg*,*end*).
- Алгоритм `is_sorted()_until` возвращает позицию первого элемента в диапазоне [*beg*,*end*), нарушающего порядок в диапазоне, или *end*, если порядок не нарушается.
- Первая и третья формы для сравнения элементов используют оператор <. Вторая и четвертая формы используют бинарный предикат *op*(*elem1*,*elem2*), который должен возвращать значение `true`, если элемент *elem1* меньше, чем элемент *elem2*.
- Если диапазон пустой или содержит только один элемент, алгоритмы возвращают значение `true` или позицию *end* соответственно.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Эти алгоритмы доступны начиная со стандарта C++11.
- Сложность: линейная (не более *numElems*−1 вызовов оператора < или предиката *op*()).

Следующая программа демонстрирует использование этих алгоритмов:

```
// algo/issorted1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll1 = { 1, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

   PRINT_ELEMENTS(coll1,"coll1: ");

   // проверяем, упорядочена ли коллекция coll1
   if (is_sorted (coll1.begin(), coll1.end())) {
       cout << "coll1 is sorted" << endl;
   }
   else {
       cout << "coll1 is not sorted" << endl;
   }
```

```
    map<int,string> coll2;
    coll2 = { {1,"Bill"}, {2,"Jim"}, {3,"Nico"}, {4,"Liu"}, {5,"Ai"} };
    PRINT_MAPPED_ELEMENTS(coll2,"coll2: ");

    // определяем предикат для сравнения имен
    auto compareName = [](const pair<int,string>& e1,
                          const pair<int,string>& e2){
    return e1.second<e2.second;
};

    // проверяем, упорядочены ли имена в коллекции coll2
    if (is_sorted (coll2.cbegin(), coll2.cend(),
                   compareName)) {
        cout << "names in coll2 are sorted" << endl;
    }
    else {
        cout << "names in coll2 are not sorted" << endl;
    }

    // выводим первое имя, нарушающее порядок
    auto pos = is_sorted_until (coll2.cbegin(), coll2.cend(),
                                compareName);
    if (pos != coll2.end()) {
        cout << "first unsorted name: " << pos->second << endl;
    }
}
```

Программа выводит следующий результат:

```
coll1: 1 1 2 3 4 5 6 7 8 9
coll1 is sorted
coll2: [1,Bill] [2,Jim] [3,Nico] [4,Liu] [5,Ai]
names in coll2 are not sorted
first unsorted name: Liu
```

Отметим, что алгоритм `is_sorted_until()` возвращает позицию первого неупорядоченного элемента в виде итератора, поэтому для доступа к имени мы должны выполнять операцию `pos->second` (значение в паре "ключ–значение").

Проверка разделения

```
bool
is_partitioned (InputIterator beg, InputIterator end, UnaryPredicate op)

ForwardIterator
partition_point (ForwardIterator beg, ForwardIterator end,
UnaryPredicate op)
```

- Алгоритм `is_partitioned()` проверяет, разделены ли элементы в диапазоне [*beg*,*end*), так что все элементы, удовлетворяющие предикату *op*(), расположены перед элементами, которые этому предикату не удовлетворяют.

- Алгоритм `partition_point()` возвращает позицию первого элемента в *разделенном* диапазоне [*beg*,*end*). Следовательно, для диапазона [*beg*,*end*) алгоритм `is_partitioned()` должен возвращать значение `true`.
- Алгоритмы используют унарный предикат *op*(*elem*), который должен возвращать значение `true`, если элемент *elem* предшествует точке разделения (т.е. аргумент `elem` меньше, чем все элементы, для которых предикат возвращает значение `false`).
- Если диапазон пустой, то алгоритм `partition_point()` возвращает позицию *end*.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Эти алгоритмы доступны начиная со стандарта C++11.
- Сложность:
  - алгоритм `is_partitioned()`: линейная (не больше *numElems* вызовов предиката *op*());
  - алгоритм `partition_point()`: логарифмическая для итераторов произвольного доступа и линейная в других случаях (в любом случае не больше log(*numElems*) вызовов предиката *op*()).

Следующая программа демонстрирует использование этих алгоритмов:

```
// algo/ispartitioned1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll = { 5, 3, 9, 1, 3, 4, 8, 2, 6 };

   PRINT_ELEMENTS(coll,"coll: ");

   // определяем предикат: проверяем нечетность элемента:
   auto isOdd = [](int elem) {
                   return elem%2==1;
                };

   // проверяем, разделена ли коллекция coll на нечетные и четные элементы
   if (is_partitioned (coll.cbegin(), coll.cend(), // диапазон
                       isOdd)) {                   // предикат
       cout << "coll is partitioned" << endl;

      // ищем первый четный элемент:
      auto pos = partition_point (coll.cbegin(),coll.cend(),
                                  isOdd);
      cout << "first even element: " << *pos << endl;
   }
   else {
      cout << "coll is not partitioned" << endl;
   }
}
```

Программа выводит следующий результат:

```
coll: 5 3 9 1 3 4 8 2 6
coll is partitioned
first even element: 4
```

Проверка того, что элементы диапазона образуют невозрастающую пирамиду

```
bool
is_heap (RandomAccessIterator beg, RandomAccessIterator end)

bool
is_heap (RandomAccessIterator beg, RandomAccessIterator end,
BinaryPredicate op)

RandomAccessIterator
is_heap_until (RandomAccessIterator beg, RandomAccessIterator end)

RandomAccessIterator
is_heap_until (RandomAccessIterator beg, RandomAccessIterator end,
               BinaryPredicate op)
```

- Алгоритм `is_heap()` проверяет, образуют ли элементы диапазона [*beg*,*end*) невозрастающую пирамиду (см. раздел 11.9.4), т.е. позиция *beg* соответствует максимальному элементу или одному из максимальных элементов.
- Алгоритм `is_heap()_until` возвращает позицию первого элемента в диапазоне [*beg*,*end*), который нарушает свойство пирамидальности (т.е. больше родительского элемента) или позицию *end*, если такого элемента нет.
- Первая и третья формы для сравнения элементов используют оператор <. Вторая и четвертая формы используют бинарный предикат *op*(*elem1*,*elem2*), который должен возвращать значение `true`, если элемент *elem1* меньше, чем элемент *elem2*.
- Если диапазон пустой или содержит только один элемент, алгоритмы возвращают значение `true` или позицию *end* соответственно.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Эти алгоритмы доступны начиная со стандарта C++11.
- Сложность: линейная (не более *numElems*-1 вызовов оператора < или предиката *op*()).

Следующая программа демонстрирует использование этих алгоритмов:

```
// algo/isheap1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
```

```
    vector<int> coll1 = { 9, 8, 7, 7, 7, 5, 4, 2, 1 };
    vector<int> coll2 = { 5, 3, 2, 1, 4, 7, 9, 8, 6 };

    PRINT_ELEMENTS(coll1,"coll1: ");
    PRINT_ELEMENTS(coll2,"coll2: ");

    // проверка, являются ли коллекции пирамидами
    cout << boolalpha << "coll1 is heap: "
         << is_heap (coll1.cbegin(), coll1.cend()) << endl;
    cout << "coll2 is heap: "
         << is_heap (coll2.cbegin(), coll2.cend()) << endl;

    // выводим первый элемент, нарушающий порядок пирамиды в коллекции coll2
    auto pos = is_heap_until (coll2.cbegin(), coll2.cend());
    if (pos != coll2.end()) {
        cout << "first non-heap element: " << *pos << endl;
    }
}
```

Программа выводит следующий результат:

```
coll1: 9 8 7 7 7 5 4 2 1
coll2: 5 3 2 1 4 7 9 8 6
coll1 is heap: true
coll2 is heap: false
first non-heap element: 4
```

Все, хотя бы один или ни одного

```
bool
all_of (InputIterator beg, InputIterator end, UnaryPredicate op)
```

```
bool
any_of (InputIterator beg, InputIterator end, UnaryPredicate op)
```

```
bool
none_of (InputIterator beg, InputIterator end, UnaryPredicate op)
```

- Эти алгоритмы возвращают результат проверки того, что унарный предикат *op* (*elem*) возвращает значение `true` для всех, хотя бы для одного или ни для одного из элементов диапазона [*beg*,*end*).
- Если диапазон пуст, то алгоритмы `all_of()` и `none_of()` возвращают значение `true`, в то время как алгоритм `any_of()` возвращает значение `false`.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Эти алгоритмы доступны начиная со стандарта C++11.
- Сложность: линейная (не больше *numElems* вызовов предиката *op*()).

Использование этих алгоритмов демонстрирует следующая программа:

```
// algo/allanynone1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll;
   vector<int>::iterator pos;

   INSERT_ELEMENTS(coll,1,9);
   PRINT_ELEMENTS(coll,"coll: ");

   // определяем объект для предиката (используя лямбда-функцию)
   auto isEven = [](int elem) {
                    return elem%2==0;
                 };

   // выводим все, хотя бы один или ни одного четного элемента
   cout << boolalpha << "all even?: "
        << all_of(coll.cbegin(),coll.cend(), isEven) << endl;
   cout << "any even?: "
        << any_of(coll.cbegin(),coll.cend(), isEven) << endl;
   cout << "none even?: "
        << none_of(coll.cbegin(),coll.cend(), isEven) << endl;
}
```

Программа выводит следующий результат:

```
coll: 1 2 3 4 5 6 7 8 9
all even?: false
any even?: true
none even?: false
```

11.6. Модифицирующие алгоритмы

В разделе описываются алгоритмы, изменяющие элементы диапазона. Существуют два способа модификации элементов.

1. Изменить их непосредственно, обходя последовательность.
2. Изменить их, копируя из диапазона-источника в диапазон-получатель.

Несколько модифицирующих алгоритмов обеспечивают оба способа модификации элементов в диапазоне. В этом случае к их имени добавляется суффикс `_copy`.

В качестве диапазона-получателя нельзя использовать ассоциативный или неупорядоченный контейнер, поскольку элементы в этих контейнерах являются константными. Если бы это было возможно, то можно было бы испортить автоматическую сортировку или хеширование.

Все алгоритмы, имеющие отдельный диапазон-получатель, возвращают позицию, находящуюся после последнего скопированного элемента этого диапазона.

11.6.1. Копирование элементов

```
OutputIterator
copy (InputIterator sourceBeg, InputIterator sourceEnd,
      OutputIterator destBeg)

OutputIterator
copy_if (InputIterator sourceBeg, InputIterator sourceEnd,
         OutputIterator destBeg,
         UnaryPredicate op)

OutputIterator
copy_n (InputIterator sourceBeg,
        Size num,
        OutputIterator destBeg)

BidirectionalIterator2
copy_backward (BidirectionalIterator1 sourceBeg,
               BidirectionalIterator1 sourceEnd,
               BidirectionalIterator2 destEnd)
```

- Все алгоритмы копируют все элементы диапазона-источника ([*sourceBeg*,*sourceEnd*) или *num* элементов, начиная с позиции *sourceBeg*) в диапазон-получатель, начиная с позиции *destBeg* или заканчивая позицией *destEnd* соответственно.
- Они возвращают позицию, находящуюся после последнего скопированного элемента в диапазоне-получателе (позицию первого не перезаписанного элемента).
- При работе с алгоритмом `copy()` позиция *destBeg* не должна быть частью диапазона [*sourceBeg*,*sourceEnd*). При работе с алгоритмом `copy_if()` диапазон-источник и диапазон-получатель не должны перекрываться. При работе с алгоритмом `copy_backward()` позиция *destEnd* не должна быть частью диапазона (*sourceBeg*,*sourceEnd*].
- Алгоритм `copy()` выполняет прямой обход последовательности, в то время как алгоритм `copy_backward()` выполняет обратный обход. Эта разница имеет значение, только если диапазон-источник и диапазон-получатель перекрываются.
- Для копирования подынтервала в начало следует использовать алгоритм `copy()`. В таком случае при работе с алгоритмом `copy()` позиция *destBeg* должна находиться до позиции *sourceBeg*.
- Для копирования подынтервала в конец следует использовать алгоритм `copy_backward()`. В таком случае при работе с алгоритмом `copy_backward()` позиция *destEnd* должна находиться за позицией *sourceEnd*.
- Итак, если третий аргумент является элементом диапазона-источника, заданного двумя первыми аргументами, следует использовать другой алгоритм. Отметим, что переключение на другой алгоритм означает, что вместо начала диапазона вы передаете алгоритму его конец.
- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Реализация алгоритма `copy()` приведена в разделе 9.4.2.

- С появлением стандарта C++11, если элементы источника больше не используются, следует отдать предпочтение алгоритму `move()` по сравнению с алгоритмом `copy()` и алгоритму `move_backward()` по сравнению с алгоритмом `copy_backward()` (см. раздел 11.6.2).
- До появления стандарта C++11 алгоритмов `copy_if()` и `copy_n()` не было. Для копирования элементов, удовлетворяющих определенному критерию, приходилось использовать алгоритм `remove_copy_if()` (см. раздел 11.7.1) с предикатом отрицания.
- Для того чтобы при копировании элементы изменили порядок следования на противоположный, следует использовать алгоритм `reverse_copy()` (см. раздел 11.8.1). Отметим, что алгоритм `reverse_copy()` может оказаться чуть более эффективным, чем алгоритм `copy()` с обратными итераторами.
- Для присвоения всех элементов контейнера следует использовать оператор присваивания (см. раздел 8.4), если контейнеры имеют одинаковый тип; или функцию-член `assign()`, если контейнеры имеют разные типы (см. раздел 8.4).
- Для удаления элементов при их копировании следует использовать алгоритмы `remove_copy()` и `remove_copy_if()` (см. раздел 11.7.1).
- Для изменения элементов в процессе копирования следует использовать алгоритм `transform()` (см. раздел 11.6.3) или `replace_copy()` (см. раздел 11.6.6).
- Для копирования элементов в два диапазона-получателя, удовлетворяющих и не удовлетворяющих заданному предикату, следует использовать алгоритм `partition_copy()` (см. раздел 11.8.6).
- Сложность: линейная (*numElems* присваиваний).

Следующий пример демонстрирует простые вызовы алгоритма `copy()` (соответствующий вариант с использованием алгоритма `move()` приведен в разделе 11.6.2):

```
// algo/copy1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<string> coll1 = { "Hello", "this", "is", "an", "example" };
   list<string> coll2;

   // копируем элементы из контейнера coll1 в контейнер coll2
   // - используем итератор вставки, чтобы выполнить вставку, а не замену

   copy (coll1.cbegin(), coll1.cend(), // источник
         back_inserter(coll2));        // получатель

   // выводим элементы контейнера coll2
   // - копируем элементы с помощью итератора ostream
   copy (coll2.cbegin(), coll2.cend(),        // источник
         ostream_iterator<string>(cout," ")); // получатель
   cout << endl;
```

```
   // копируем элементы контейнера coll1 в контейнер coll2 в обратном порядке
   // - теперь заменяя исходные элементы копируемыми
   copy (coll1.crbegin(), coll1.crend(),      // источник
         coll2.begin());                      // получатель

   // снова выводим элементы контейнера coll2
   copy (coll2.cbegin(), coll2.cend(),        // источник
         ostream_iterator<string>(cout," ")); // получатель
   cout << endl;
}
```

В этом примере для вставки элементов в диапазон-получатель используются итераторы вставки в конец (см. раздел 9.4.2). Без использования итераторов вставки алгоритм `copy()` перезаписывал бы пустую коллекцию `coll2`, что привело бы к непредсказуемым последствиям. Аналогично в примере используется итераторы `ostream` (см. раздел 9.4.3) для использования стандартного потока вывода в качестве получателя. Программа выводит следующий результат:

```
Hello this is an example
example an is this Hello
```

Следующий пример демонстрирует различие между алгоритмами `copy()` и `copy_backward()`:

```
// algo/copy2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   // инициализируем источник строкой ".....abcdef....."
   vector<char> source(10,'.');
   for (int c='a'; c<='f'; c++) {
      source.push_back(c);
   }
   source.insert(source.end(),10,'.');
   PRINT_ELEMENTS(source,"source: ");

   // копируем все трехбуквенные элементы в позицию перед буквой 'a'
   vector<char> c1(source.cbegin(),source.cend());
                copy (c1.cbegin()+10, c1.cbegin()+16, // источник
                c1.begin()+7);                        // получатель
   PRINT_ELEMENTS(c1,"c1:     ");

   // копируем все трехбуквенные элементы в позицию после буквы 'f'
   vector<char> c2(source.cbegin(),source.cend());
               copy_backward (c2.cbegin()+10, c2.cbegin()+16, // источник
               c2.begin()+19);                                // получатель
   PRINT_ELEMENTS(c2,"c2:     ");
}
```

Отметим, что в обоих вызовах алгоритмов `copy()` и `copy_backward()` третий аргумент не является частью диапазона-источника. Программа выводит следующий результат:

```
source: . . . . . a b c d e f . . . . . .
c1:     . . . . . a b c d e f d e f . . . . . .
c2:     . . . . . a b c a b c d e f . . . . . .
```

Третий пример демонстрирует использование алгоритма `copy()` в качестве фильтра данных между стандартными потоками ввода и вывода. Программа читает слова и выводит их на экран по одному в строке.

```
// algo/copy3.cpp

#include <iostream>
#include <algorithm>
#include <iterator>
#include <string>
using namespace std;

int main()
{
   copy (istream_iterator<string>(cin),          // начало источника
         istream_iterator<string>(),             // конец источника
         ostream_iterator<string>(cout,"\n"));   // получатель
}
```

11.6.2. Перемещение элементов

```
OutputIterator
move (InputIterator sourceBeg, InputIterator sourceEnd,
      OutputIterator destBeg)

BidirectionalIterator2
move_backward (BidirectionalIterator1 sourceBeg,
               BidirectionalIterator1 sourceEnd,
               BidirectionalIterator2 destEnd)
```

- Оба алгоритма перемещают все элементы из диапазона-источника [*sourceBeg*,*sourceEnd*) в диапазон-получатель, начинающийся с позиции *destBeg* или заканчивающийся позицией *destEnd* соответственно.
- Для каждого элемента осуществляется следующий вызов:

  ```
  *destElem=std::move(*sourceElem)
  ```

- Таким образом, если тип элемента предусматривает семантику перемещения, то значения элементов источника становятся неопределенными, так что элемент источника больше не должен использоваться, за исключением повторной инициализации или присвоения ему нового значения. В противном случае элементы копируются так же, как и с помощью алгоритма `copy()` или `copy_backward()` (см. раздел 11.6.1).
- Алгоритмы возвращают позицию, находящуюся за последним скопированным элементом в диапазоне-получателе (первый элемент, который не заменяется).

- При работе с алгоритмом `move()` позиция *destBeg* не должна быть частью диапазона [*sourceBeg*,*sourceEnd*). При работе с алгоритмом `move_backward()` позиция *destEnd* не должна быть частью диапазона (*sourceBeg*,*sourceEnd*].
- Алгоритм `move()` выполняет прямой обход последовательности, в то время как алгоритм `move_backward()` выполняет обратный обход. Эта разница имеет значение, только если диапазон-источник и диапазон-получатель перекрываются.
  - Для перемещения подынтервала в начало следует использовать алгоритм `move()`. Таким образом, при работе с алгоритмом `move()` позиция *destBeg* должна находиться до позиции *sourceBeg*.
  - Для перемещения подынтервала в конец следует использовать алгоритм `move_backward()`. Таким образом, при работе с алгоритмом `move_backward()` позиция *destEnd* должна находиться после позиции *sourceEnd*.
- Итак, если третий аргумент является элементом диапазона-источника, заданного двумя первыми аргументами, следует использовать другой алгоритм. Отметим, что переключение на другой алгоритм означает, что алгоритму вместо начала диапазона передается его конец. Пример, иллюстрирующий эту разницу, приведен в разделе 11.6.1.
- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Эти алгоритмы доступны начиная со стандарта C++11.
- Сложность: линейная (*numElems* перемещающих присваиваний).

Следующий пример демонстрирует простые вызовы алгоритма `move()`. Он представляет собой усовершенствованный вариант примера `algo/copy1.cpp` (см. раздел 11.6.1), в котором везде, где возможно, вместо алгоритма `copy()` используется алгоритм `move()`:

```
// algo/move1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<string> coll1 = { "Hello", "this", "is", "an", "example" };
   list<string> coll2;

   // копируем элементы из контейнера coll1 в контейнер coll2
   // - используем итератор вставки в конец, чтобы вставлять, а не заменять
   // - используем алгоритм copy(), потому что элементы в контейнере coll1
   // используются снова
   copy (coll1.cbegin(), coll1.cend(), // источник
         back_inserter(coll2));        // получатель

   // выводим элементы контейнера coll2
   // - копируем элемент в поток cout, используя итератор ostream
   // - используем алгоритм move(), потому что эти элементы в контейнере coll2
   //   больше не используются
   move (coll2.cbegin(), coll2.cend(),          // источник
         ostream_iterator<string>(cout," "));   // получатель
   cout << endl;
```

```
    // копируем элементы контейнера coll1 в контейнер coll2 в обратном порядке
    // - с перезаписью (размер coll2.size() достаточен)
    // - используем алгоритм move(), потому что элементы в контейнере coll1
    //   больше не используются
    move (coll1.crbegin(), coll1.crend(),      // источник
          coll2.begin());                      // получатель

    // снова выводим элементы контейнера coll2
    // - используем алгоритм move(), потому что элементы контейнера coll2
    //   больше не используются
    move (coll2.cbegin(), coll2.cend(),        // источник
          ostream_iterator<string>(cout," "));  // получатель
    cout << endl;
}
```

Отметим, что элементы в контейнере `coll2` после первого вывода остаются в неопределенном состоянии, потому что используется алгоритм `move()`. Однако размер контейнера `coll2` все еще равен 5, поэтому мы можем заменить эти элементы при втором вызове алгоритма `move()`. Программа выдает следующий результат:

```
Hello this is an example
example an is this Hello
```

11.6.3. Преобразование и объединение элементов

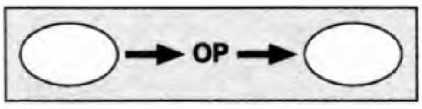
Алгоритм `transform()` имеет две формы.

1. Первая форма имеет четыре аргумента. Она преобразовывает элементы из источника в диапазон-получатель. Следовательно, эта форма копирует и модифицирует элементы за один этап.
2. Вторая форма имеет пять аргументов. Она объединяет элементы двух диапазонов-источников и записывает результаты в диапазон-получатель.

Преобразование элементов

```
OutputIterator
transform (InputIterator sourceBeg, InputIterator sourceEnd,
           OutputIterator destBeg,
           UnaryFunc op)
```

- Вызывает предикат *op*(*elem*) для каждого элемента в диапазоне-источнике [*sourceBeg*,*sourceEnd*) и записывает результат каждого вызова предиката *op* в диапазон-получатель, начиная с позиции *destBeg*.



- Возвращает позицию после последнего преобразованного элемента в диапазоне-получателе (позиция первого не перезаписанного элемента).

- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Позиции *sourceBeg* и *destBeg* могут совпадать. Следовательно, как и алгоритм `for_each()`, этот алгоритм можно использовать для модификации элементов в последовательности. См. сравнение с алгоритмом `for_each()` в разделе 11.2.2.
- Для замены элементов, соответствующих критериям с конкретным значением, следует использовать алгоритм `replace()` (см. раздел 11.6.6).
- Сложность: линейная (*numElems* вызовов предиката *op*()).

Следующая программа демонстрирует использование алгоритма `transform()`:

```
// algo/transform1.cpp

#include "algostuff.hpp"
using namespace std;
using namespace std::placeholders;

int main()
{
   vector<int> coll1;
   list<int> coll2;

   INSERT_ELEMENTS(coll1,1,9);
   PRINT_ELEMENTS(coll1,"coll1: ");

   // изменяем знак всех элементов в контейнере coll1
   transform (coll1.cbegin(), coll1.cend(), // источник
              coll1.begin(),                // получатель
              negate<int>());               // операция

   PRINT_ELEMENTS(coll1,"negated: ");

   // преобразовываем элемент контейнера coll1 в контейнер coll2,
   // увеличивая их значения в десять раз
   transform (coll1.cbegin(), coll1.cend(),    // источник
              back_inserter(coll2),            // получатель
              bind(multiplies<int>(),_1,10));  // операция

   PRINT_ELEMENTS(coll2,"coll2: ");

   // выводим элементы контейнера coll2 с противоположными знаками
   // и в обратном порядке
   transform (coll2.crbegin(), coll2.crend(),   // источник
              ostream_iterator<int>(cout," "),  // диапазон
              [](int elem){                     // операция
                 return -elem;
              });
   cout << endl;
}
```

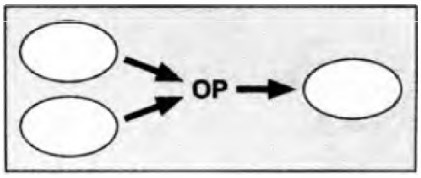
Программа выводит следующий результат:

```
coll1: 1 2 3 4 5 6 7 8 9
negated: -1 -2 -3 -4 -5 -6 -7 -8 -9
coll2: -10 -20 -30 -40 -50 -60 -70 -80 -90
90 80 70 60 50 40 30 20 10
```

Объединение элементов двух последовательностей

```
OutputIterator
transform (InputIterator1 source1Beg, InputIterator1 source1End,
           InputIterator2 source2Beg,
           OutputIterator destBeg,
           BinaryFunc op)
```

- Вызывает предикат *op* (*source1Elem*, *source2Elem*) ко всем соответствующим элементам из первого диапазона-источника [*source1Beg*,*source1End*) и второго диапазона-источника, начиная с позиции *source2Beg*, и записывает результат каждого вызова в диапазон-получатель, начиная с позиции *destBeg*.



- Возвращает позицию после последнего преобразованного элемента в диапазоне-получателе (позицию первого не перезаписанного элемента).
- Вызывающая сторона должна гарантировать, что второй диапазон-источник достаточно большой (имеет как минимум столько же элементов, сколько и первый).
- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Позиции *sourceBeg*, *source2Beg* и *destBeg* могут совпадать. Следовательно, как и алгоритм `for_each()`, этот алгоритм можно использовать для комбинирования элементов и переписывать исходные элементы результатами вычислений.
- Сложность: линейная (*numElems* вызовов предиката *op*()).

Использование этой формы алгоритма `transform()` демонстрирует следующая программа:

```
// algo/transform2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll1;
   list<int> coll2;

   INSERT_ELEMENTS(coll1,1,9);
   PRINT_ELEMENTS(coll1,"coll1:   ");
```

```
   // возводим каждый элемент в квадрат
   transform (coll1.cbegin(), coll1.cend(), // первый диапазон-источник
              coll1.cbegin(),               // второй диапазон-источник
              coll1.begin(),                // диапазон-получатель
              multiplies<int>());           // операция
   PRINT_ELEMENTS(coll1,"squared: ");

   // Суммируем каждый элемент в порядке прямого обхода с каждым элементом
в порядке
   // обратного обхода и вставляем результат в контейнер coll2
   transform (coll1.cbegin(), coll1.cend(), // первый диапазон-источник
              coll1.crbegin(),              // второй диапазон-источник
              back_inserter(coll2),         // диапазон-получатель
              plus<int>());                 // операция
   PRINT_ELEMENTS(coll2,"coll2:   ");

   // выводим разности между двумя соответствующими элементами
   cout << "diff:    ";
   transform (coll1.cbegin(), coll1.cend(),     // первый диапазон-источник
              coll2.cbegin(),                   // второй диапазон-источник
              ostream_iterator<int>(cout, " "), // диапазон-получатель
              minus<int>());                    // операция
   cout << endl;
}
```

Программа выводит следующий результат:

```
coll1:   1 2 3 4 5 6 7 8 9
squared: 1 4 9 16 25 36 49 64 81
coll2:   82 68 58 52 50 52 58 68 82
diff:    -81 -64 -49 -36 -25 -16 -9 -4 -1
```

11.6.4. Обмен элементов

```
ForwardIterator2
swap_ranges (ForwardIterator1 beg1, ForwardIterator1 end1,
             ForwardIterator2 beg2)
```

- Обменивает элементы диапазона [*beg1*,*end1*) с соответствующими элементами диапазона, начинающегося с позиции *beg2*.
- Возвращает позицию после последнего переставленного элемента во втором диапазоне.
- Вызывающая сторона должна гарантировать, что второй диапазон достаточно большой.
- Оба диапазона не должны перекрываться.
- Для перестановки всех элементов контейнеров одного и того же типа следует использовать функцию-член `swap()`, потому что эта функция обычно имеет константную сложность (см. раздел 8.4).
- Сложность: линейная (*numElems* перестановок).

Следующий пример демонстрирует использование алгоритма `swap_ranges()`:

```
// algo/swapranges1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll1;
   deque<int> coll2;

   INSERT_ELEMENTS(coll1,1,9);
   INSERT_ELEMENTS(coll2,11,23);

   PRINT_ELEMENTS(coll1,"coll1: ");
   PRINT_ELEMENTS(coll2,"coll2: ");

   // обмен элементов контейнера coll1 с соответствующими элементами
   //контейнера coll2
   deque<int>::iterator pos;
   pos = swap_ranges (coll1.begin(), coll1.end(),    // первый диапазон
                      coll2.begin());                // второй диапазон

   PRINT_ELEMENTS(coll1,"\ncoll1: ");
   PRINT_ELEMENTS(coll2,"coll2: ");

   if (pos != coll2.end()) {
       cout << "first element not modified: "
            << *pos << endl;
   }

   // зеркально отображаем три первых и три последних элемента контейнера coll2
   swap_ranges (coll2.begin(), coll2.begin()+3,      // первый диапазон
                coll2.rbegin());                     // второй диапазон

   PRINT_ELEMENTS(coll2,"\ncoll2: ");
}
```

Первый вызов алгоритма `swap_ranges()` обменивает элементы контейнера `coll1` с соответствующими элементами контейнера `coll2`. Остальные элементы контейнера `coll2` не модифицируются. Алгоритм `swap_ranges()` возвращает позицию первого немодифицированного элемента. Второй вызов обменивает первые и последние три элемента контейнера `coll2`. Один из итераторов является обратным, так что элементы отображаются зеркально (обмениваются извне вовнутрь). Программа выводит следующий результат:

```
coll1: 1 2 3 4 5 6 7 8 9
coll2: 11 12 13 14 15 16 17 18 19 20 21 22 23

coll1: 11 12 13 14 15 16 17 18 19
coll2: 1 2 3 4 5 6 7 8 9 20 21 22 23
first element not modified: 20

coll2: 23 22 21 4 5 6 7 8 9 20 3 2 1
```

11.6.5. Присвоение новых значений

Заполнение одним и тем же значением

OutputIterator
fill (ForwardIterator *beg*, ForwardIterator *end*,
const T& *newValue*)

OutputIterator
fill_n (OutputIterator *beg*, Size *num*,
const T& *newValue*)

- Алгоритм `fill()` присваивает значение *newValue* каждому элементу в диапазоне [*beg*,*end*).
- Алгоритм `fill_n()` присваивает значение *newValue* первым *num* элементам в диапазоне, начинающемся с позиции *beg*. Если величина *num* отрицательная, то алгоритм `fill_n()` ничего не делает (в соответствии со стандартом C++11).
- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Начиная со стандарта C++11 алгоритм `fill_n()` возвращает позицию, находящуюся после последнего модифицированного элемента (*beg*+*num*), или позицию *beg*, если значение *num* отрицательное (до появления стандарта C++11 алгоритм `fill_n()` имел тип возвращаемого значения `void`).
- Сложность: линейная (*numElems*, *num* или 0 присваиваний).

Следующая программа демонстрирует использование алгоритмов `fill()` и `fill_n()`:

```
// algo/fill1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   // десять раз выводим 7.7
   fill_n(ostream_iterator<float>(cout, " "), // начало диапазона-получателя
          10,                                 // количество
          7.7);                               // новое значение
   cout << endl;

   list<string> coll;

   // девять раз вставляем строку "hello"
   fill_n(back_inserter(coll),                // начало диапазона-получателя
          9,                                  // количество
          "hello");                           // новое значение

   PRINT_ELEMENTS(coll,"coll: ");

   // заменяем все элементы строкой "again"
   fill(coll.begin(), coll.end(),             // получатель
```

```
        "again");                              // новое значение

    PRINT_ELEMENTS(coll,"coll: ");

    // заменяем все, кроме двух элементов, строкой "hi"
    fill_n(coll.begin(),                       // начало диапазона-получателя
           coll.size()-2,                      // количество
           "hi");                              // новое значение

    PRINT_ELEMENTS(coll,"coll: ");

    // заменяем элементы от второго до предпоследнего строкой "hmmm"
    list<string>::iterator pos1, pos2;
    pos1 = coll.begin();
    pos2 = coll.end();
    fill (++pos1, --pos2,                      // получатель
          "hmmm");                             // новое значение
    PRINT_ELEMENTS(coll,"coll: ");
}
```

Первый вызов показывает, как использовать алгоритм `fill_n()` для вывода определенного количества значений. Другие вызовы алгоритмов `fill()` и `fill_n()` вставляют и заменяют значения в списке строк. Программа выводит следующий результат:

```
7.7 7.7 7.7 7.7 7.7 7.7 7.7 7.7 7.7 7.7
coll: hello hello hello hello hello hello hello hello hello
coll: again again again again again again again again again
coll: hi hi hi hi hi hi hi again again
coll: hi hmmm hmmm hmmm hmmm hmmm hmmm hmmm again
```

Присвоение сгенерированных значений

OutputIterator
generate (ForwardIterator *beg*, ForwardIterator *end*,
Func *op*)

OutputIterator
generate_n (OutputIterator *beg*, Size *num*,
Func *op*)

- Алгоритм `generate()` присваивает значения, сгенерированные вызовом *op*() для каждого элемента в диапазоне [*beg*,*end*).
- Алгоритм `generate_n()` присваивает значения, сгенерированные вызовом *op*(), первым *num* элементам в диапазоне, начинающемся с позиции *beg*. Если значение num отрицательное, то алгоритм `generate_n()` ничего не делает (начиная со стандарта C++11).
- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Начиная со стандарта C++11 алгоритм `generate_n()` возвращает позицию, расположенную после последнего модифицированного элемента (*beg*+*num*), или

позицию *beg*, если значение *num* отрицательное (до появления стандарта C++11 алгоритм `generate_n()` имел тип возвращаемого значения `void`).

- Сложность: линейная (*numElems*, *num* или 0 вызовов операции *op*() и присваиваний).

Следующая программа демонстрирует использование алгоритмов `generate()` и `generate_n()` для вставки или присваивания случайных чисел:

```
// algo/generate1.cpp

#include <cstdlib>
#include "algostuff.hpp"
using namespace std;

int main()
{
   list<int> coll;

   // вставляем пять случайных элементов
   generate_n (back_inserter(coll),     // начало диапазона-получателя
               5,                       // количество
               rand);                   // генератор нового значения
   PRINT_ELEMENTS(coll);

   // заменяем элементы пятью случайными числами
   generate (coll.begin(), coll.end(),  // диапазон-получатель
             rand);                     // генератор нового значения
   PRINT_ELEMENTS(coll);
}
```

Функция `rand()` описана в разделе 17.3. Программа может выводить следующий результат:

```
1481765933 1085377743 1270216262 1191391529 812669700
553475508 445349752 1344887256 730417256 1812158119
```

Результат зависит от платформы, потому что последовательность случайных чисел, генерируемых функцией `rand()`, не стандартизирована.

Пример, демонстрирующий использование алгоритма `generate()` и функциональных объектов для генерирования последовательности чисел, приведен в разделе 10.1.2.

Присвоение последовательности возрастающих чисел

```
void
iota (ForwardIterator beg, ForwardIterator end,
      T startValue)
```

- Присваивает значения *startValue*, *startValue*+1, *startValue*+2 и т.д.
- Предусмотрен стандартом C++11.
- Сложность: линейная (*numElems* присваиваний и увеличений).

Следующая программа демонстрирует использование алгоритма `iota()`:

```
// algo/iota1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   array<int,10> coll;

   iota (coll.begin(), coll.end(), // диапазон-получатель
         42);                      // начальное значение
   PRINT_ELEMENTS(coll,"coll: ");
}
```

Программа выводит следующий результат:

```
coll: 42 43 44 45 46 47 48 49 50 51
```

11.6.6. Замена элементов

Замена значений в последовательности

```
void
replace (ForwardIterator beg, ForwardIterator end,
         const T& oldValue, const T& newValue)

void
replace_if (ForwardIterator beg, ForwardIterator end,
            UnaryPredicate op, const T& newValue)
```

- Алгоритм `replace()` заменяет значением *newValue* каждый элемент диапазона [*beg*,*end*), равный *oldValue*.
- Алгоритм `replace_if()` заменяет значением *newValue* каждый элемент диапазона [*beg*,*end*), для которого унарный предикат *op*(*elem*) возвращает значение `true`.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Сложность: линейная (*numElems* сравнений или вызовов предиката *op*() соответственно).

Следующая программа демонстрирует использование алгоритмов `replace()` и `replace_if()`:

```
// algo/replace1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   list<int> coll;
```

```
    INSERT_ELEMENTS(coll,2,7);
    INSERT_ELEMENTS(coll,4,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // заменяем числом 42 все элементы, равные 6
    replace (coll.begin(), coll.end(),     // диапазон
             6,                            // старое значение
             42);                          // новое значение
    PRINT_ELEMENTS(coll,"coll: ");

    // заменяем нулем все элементы, которые меньше 5
    replace_if (coll.begin(), coll.end(), // диапазон
                [](int elem){             // критерий замены
                   return elem<5;
                },
                0);                       // новое значение
    PRINT_ELEMENTS(coll,"coll: ");
}
```

Программа выводит следующие результаты:

```
coll: 2 3 4 5 6 7 4 5 6 7 8 9
coll: 2 3 4 5 42 7 4 5 42 7 8 9
coll: 0 0 0 5 42 7 0 5 42 7 8 9
```

Копирование и замена элементов

```
OutputIterator
replace_copy (InputIterator sourceBeg, InputIterator sourceEnd,
              OutputIterator destBeg,
              const T& oldValue, const T& newValue)

OutputIterator
replace_copy_if (InputIterator sourceBeg, InputIterator sourceEnd,
                 OutputIterator destBeg,
                 UnaryPredicate op, const T& newValue)
```

- Алгоритм `replace_copy()` является комбинацией алгоритмов `copy()` и `replace()`. Он заменяет значением *newValue* каждый элемент диапазона-источника [*sourceBeg*,*sourceEnd*), равный значению *oldValue*, и копирует эти элементы в диапазон-получатель, начинающийся с позиции *destBeg*.
- Алгоритм `replace_copy_if()` является комбинацией алгоритмов `copy()` и `replace_if()`. Он заменяет значением *newValue* каждый элемент диапазона-источника [*sourceBeg*,*sourceEnd*), для которого унарный предикат *op*(*elem*) возвращает значение `true`, и копирует эти элементы в диапазон-получатель, начинающийся с позиции *destBeg*.
- Оба алгоритма возвращают позицию, находящуюся за последним скопированным элементом в диапазоне-получателе (позиция первого элемента, который не был заменен).
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).

- Сложность: линейная (*numElems* сравнений или вызовов предиката *op*() соответственно).

Следующая программа демонстрирует использование алгоритмов `replace_copy()` и `replace_copy_if()`:

```
// algo/replace2.cpp

#include "algostuff.hpp"
using namespace std;
using namespace std::placeholders;

int main()
{
   list<int> coll;

   INSERT_ELEMENTS(coll,2,6);
   INSERT_ELEMENTS(coll,4,9);
   PRINT_ELEMENTS(coll);

   // выводим все элементы со значением 5, заменяя их значением 55
   replace_copy(coll.cbegin(), coll.cend(),           // источник
                ostream_iterator<int>(cout," "),      // получатель
                5,                                    // старое значение
                55);                                  // новое значение
   cout << endl;

   // выводим все элементы со значением меньше 5, заменяя их значением 42
   replace_copy_if(coll.cbegin(), coll.cend(),        // источник
                   ostream_iterator<int>(cout," "),   // получатель
                   bind(less<int>(),_1,5),            // критерий замены
                   42);                               // новое значение
    cout << endl;

   // выводим все элементы, заменяя нечетные элементы нулями
   replace_copy_if(coll.cbegin(), coll.cend(),        // источник
                   ostream_iterator<int>(cout," "),   // получатель
                   [](int elem){                      // критерий замены
                      return elem%2==1;
                   },
                   0);                                // новое значение
   cout << endl;
}
```

Программа выводит следующий результат:

```
2 3 4 5 6 4 5 6 7 8 9
2 3 4 55 6 4 55 6 7 8 9
42 42 42 5 6 42 5 6 7 8 9
2 0 4 0 6 4 0 6 0 8 0
```

11.7. Алгоритмы удаления

Следующие алгоритмы удаляют элементы из диапазона в зависимости от их значения или по заданному критерию. Однако эти алгоритмы *не могут* изменить количество элементов. Они лишь логически перемещают удаляемые элементы, перезаписывая их следующими за ними. Эти алгоритмы возвращают новый логический конец диапазона (позицию, расположенную после последнего не удаленного элемента). Детали изложены в разделе 6.7.1.

11.7.1. Удаление определенных значений

Удаление элементов из последовательности

```
ForwardIterator
remove (ForwardIterator beg, ForwardIterator end,
        const T& value)

ForwardIterator
remove_if (ForwardIterator beg, ForwardIterator end,
           UnaryPredicate op)
```

- Алгоритм `remove()` удаляет все элементы диапазона [*beg*,*end*), имеющие значение *value*.
- Алгоритм `remove_if()` удаляет все элементы диапазона [*beg*,*end*), для которых унарный предикат *op*(*elem*) возвращает значение `true`.
- Оба алгоритма возвращают новый логический конец модифицированной последовательности (позицию, расположенную после последнего не удаленного элемента).
- Алгоритмы заменяют удаленные элементы следующими элементами, которые не были удалены.
- Порядок элементов, которые не были удалены, остается неизменным.
- После вызова алгоритма вызывающая сторона должна использовать новый логический конец вместо исходного конца *end* (см. раздел 6.7.1).
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Алгоритм `remove_if()` обычно копирует унарный предикат в алгоритме и использует его дважды. Это может привести к проблемам, если предикат изменяет свое состояние во время вызова функции (см. раздел 10.1.4).
- Из-за модификаций эти алгоритмы нельзя применять к ассоциативным и неупорядоченным контейнерам (см. раздел 6.7.2). Однако эти контейнеры имеют подобную функцию-член `erase()` (см. раздел 8.7.3).
- Списки имеют эквивалентную функцию-член `remove()`, обеспечивающую более высокую производительность, поскольку она изменяет ссылки, а не присваивает значения (см. раздел 8.8.1).
- Сложность: линейная (*numElems* сравнений или вызовов предиката *op*() соответственно).

Следующая программа демонстрирует использование алгоритмов `remove()` и `remove_if()`:

```
// algo/remove1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll;

   INSERT_ELEMENTS(coll,2,6);
   INSERT_ELEMENTS(coll,4,9);
   INSERT_ELEMENTS(coll,1,7);
   PRINT_ELEMENTS(coll,"coll:             ");

   // удаляем все элементы, имеющие значение 5
   vector<int>::iterator pos;
   pos = remove(coll.begin(), coll.end(),         // диапазон
                5);                               // удаляемое значение

   PRINT_ELEMENTS(coll,"size not changed: ");

   // очищаем удаленные элементы в контейнерне
   coll.erase(pos, coll.end());
   PRINT_ELEMENTS(coll,"size changed:     ");

   // удаляем все элементы, которые меньше 4
   coll.erase(remove_if(coll.begin(), coll.end(), // диапазон
                        [](int elem){             // критерий удаления
                           return elem<4;
                        }),
              coll.end());
   PRINT_ELEMENTS(coll,"<4 removed:       ");
}
```

Программа выводит следующий результат:

```
coll:             2 3 4 5 6 4 5 6 7 8 9 1 2 3 4 5 6 7
size not changed: 2 3 4 6 4 6 7 8 9 1 2 3 4 6 7 5 6 7
size changed:     2 3 4 6 4 6 7 8 9 1 2 3 4 6 7
<4 removed:       4 6 4 6 7 8 9 4 6 7
```

Удаление элементов при копировании

```
OutputIterator
remove_copy (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg,
             const T& value)

OutputIterator
remove_copy_if (InputIterator sourceBeg, InputIterator sourceEnd,
                OutputIterator destBeg,
                UnaryPredicate op)
```

- Алгоритм `remove_copy()` является комбинацией алгоритмов `copy()` и `remove()`. Он копирует каждый элемент диапазона-источника [*sourceBeg*,*sourceEnd*), не равный значению *value*, в диапазон-получатель, начинающийся с позиции *destBeg*.
- Алгоритм `remove_copy_if()` является комбинацией алгоритмов `copy()` и `remove_if()`. Он копирует каждый элемент диапазона-источника [*sourceBeg*,*sourceEnd*), для которого унарный предикат *op*(*elem*) возвращает значение `false`, в диапазон-получатель, начинающийся с позиции *destBeg*.
- Оба алгоритма возвращают позицию, находящуюся за последним скопированным элементом в диапазоне-получателе (позиция первого элемента, который не был заменен).
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Для копирования элементов в два диапазона-получателя — удовлетворяющий и не удовлетворяющий предикату — следует использовать алгоритм `partition_copy()` (см. раздел 11.8.6) (доступен начиная со стандарта C++11).
- Сложность: линейная (*numElems* сравнений или вызовов предиката *op*() соответственно).

Следующая программа демонстрирует использование алгоритмов `remove_copy()` и `remove_copy_if()`:

```
// algo/remove2.cpp

#include "algostuff.hpp"
using namespace std;
using namespace std::placeholders;

int main()
{
   list<int> coll1;

   INSERT_ELEMENTS(coll1,1,6);
   INSERT_ELEMENTS(coll1,1,9);
   PRINT_ELEMENTS(coll1);

   // выводим элементы, за исключением элементов, равных 3
   remove_copy(coll1.cbegin(), coll1.cend(),        // источник
               ostream_iterator<int>(cout," "),     // получатель
               3);                                  // удаляемое значение
   cout << endl;

   // выводим элементы, за исключением элементов, значение которых больше 4
   remove_copy_if(coll1.cbegin(), coll1.cend(),     // источник
                  ostream_iterator<int>(cout," "),  // получатель
                  [](int elem){          // критерий для элементов,
                     return elem>4;      // которые НЕ копируются
                  });
   cout << endl;
```

```
    // копируем в мультимножество все элементы, которые не меньше 4
    multiset<int> coll2;
    remove_copy_if(coll1.cbegin(), coll1.cend(),      // источник
                   inserter(coll2,coll2.end()),       // получатель
                   bind(less<int>(),_1,4));           // элементы, которые
    PRINT_ELEMENTS(coll2);                            // НЕ копируются
}
```

Программа выводит следующий результат:

```
1 2 3 4 5 6 1 2 3 4 5 6 7 8 9
1 2 4 5 6 1 2 4 5 6 7 8 9
1 2 3 4 1 2 3 4
4 4 5 5 6 6 7 8 9
```

11.7.2. Удаление дубликатов

Удаление соседних дубликатов

```
ForwardIterator
unique (ForwardIterator beg, ForwardIterator end)
```

```
ForwardIterator
unique (ForwardIterator beg, ForwardIterator end,
        BinaryPredicate op)
```

- Обе формы "сворачивают" соседние одинаковые элементы, удаляя последующие дубликаты.
- Первая форма удаляет из диапазона [*beg*,*end*) все элементы, равные предыдущим элементам. Таким образом, этот алгоритм удаляет все дубликаты, только если элементы в последовательности упорядочены или, по крайней мере, если все элементы с одинаковыми значениями являются соседними.
- Вторая форма удаляет все элементы, следующие за элементом *e*, для которых бинарный предикат *op*(*e*, *elem*) возвращает `true`. Иными словами, предикат используется для сравнения не с его предшественником, а с предыдущим элементом, который не был удален (см. следующие примеры).
- Обе формы возвращают новый логический конец модифицированной последовательности (позицию, находящуюся за последним не удаленным элементом).
- Алгоритмы перезаписывают удаленные элементы следующими за ними элементами, которые не были удалены.
- Порядок элементов, которые не были удалены, остается без изменения.
- Вызывающая сторона после вызова этого алгоритма должна использовать возвращенный новый логический конец, а не исходный конец *end* (см. раздел 6.7.1).
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Вследствие модификаций эти алгоритмы нельзя применять к ассоциативным или неупорядоченным контейнерам (см. раздел 6.7.2).

- Списки имеют эквивалентную функцию-член `unique()`, которая часто обеспечивает более высокую эффективность, потому что меняет указатели, а не присваивает значения элементов (см. раздел 8.8.1).
- Сложность: линейная (*numElems* сравнений или вызовов предиката *op*() соответственно).

Следующая программа демонстрирует использование алгоритма `unique()`:

```
// algo/unique1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   // исходные данные
   int source[] = { 1, 4, 4, 6, 1, 2, 2, 3, 1, 6, 6, 6, 5, 7,
                    5, 4, 4 };
   list<int> coll;

   // инициализируем список coll элементами из источника
   copy (begin(source), end(source),           // источник
         back_inserter(coll));                 // получатель
   PRINT_ELEMENTS(coll);

   // удаляем последовательные дубликаты
   auto pos = unique (coll.begin(), coll.end());

   // выводим не удаленные элементы
   // - используем новый логический конец
   copy (coll.begin(), pos,                    // источник
         ostream_iterator<int>(cout," "));     // получатель
   cout << "\n\n";

   // повторно инициализируем список coll элементами из источника
   copy (begin(source), end(source),           // источник
         coll.begin());                        // получатель
   PRINT_ELEMENTS(coll);

   // удаляем элемент, если ему предшествует элемент с большим значением
   coll.erase (unique (coll.begin(), coll.end(),
                       greater<int>()),
               coll.end());
   PRINT_ELEMENTS(coll);
}
```

Программа выводит следующий результат:

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

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

Первый вызов алгоритма `unique()` удаляет последовательные дубликаты. Второй вызов демонстрирует работу второй формы и удаляет все последовательные элементы, следующие за элементом, для которого сравнение с помощью предиката `greater` возвращает `true`. Например, первый элемент `6` больше следующих `1`, `2`, `2`, `3` и `1`, так что все эти элементы удаляются. Иначе говоря, предикат используется для сравнения не с предшественником, а с предыдущим элементом, который не был удален (см. описание алгоритма `unique_copy()` в следующем примере).

Удаление дубликатов при копировании

```
OutputIterator
unique_copy (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg)

OutputIterator
unique_copy (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg,
             BinaryPredicate op)
```

- Обе формы являются комбинацией алгоритмов `copy()` и `unique()`.
- Они копируют все элементы исходного диапазона [*sourceBeg*,*sourceEnd*), у которых нет дубликатов среди предыдущих элементов в диапазоне-получателе, начинающемся с позиции *destBeg*.
- Обе формы возвращают позицию после последнего скопированного элемента в целевом диапазоне (первый не перезаписанный элемент).
- Вызывающая сторона должна гарантировать, что диапазон-получатель является достаточно большим или что используются итераторы вставки.
- Сложность: линейная (*numElems* сравнений или вызовов предиката *op*() и присваиваний соответственно).

Следующая программа демонстрирует использование алгоритма `unique_copy()`:

```
// algo/unique2.cpp

#include "algostuff.hpp"
using namespace std;

bool differenceOne (int elem1, int elem2)
{
   return elem1 + 1 == elem2 || elem1 - 1 == elem2;
}

int main()
{
   // исходные данные
   int source[] = { 1, 4, 4, 6, 1, 2, 2, 3, 1, 6, 6, 6, 5, 7,
                    5, 4, 4 };

   // инициализируем список coll элементами из источника
   list<int> coll;
   copy(begin(source), end(source),                    // источник
```

```
        back_inserter(coll));                    // получатель
    PRINT_ELEMENTS(coll);

    // выводим элементы, удаляя последовательные дубликаты
    unique_copy(coll.cbegin(), coll.cend(),      // источник
                ostream_iterator<int>(cout," ")); // получатель
     cout << endl;

    // выводим элементы без последовательных элементов, отличающихся на единицу
    unique_copy(coll.cbegin(), coll.cend(),      // источник
                ostream_iterator<int>(cout," "),  // получатель
                differenceOne);                  // критерий дубликатов
    cout << endl;
}
```

Программа выводит следующий результат:

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

Отметим, что второй вызов алгоритма `unique_copy()` не удаляет элементы, отличающиеся на единицу от предыдущего элемента. Вместо этого он удаляет все элементы, отличающиеся на единицу от предыдущего элемента, *который не был удален*. Например, после трех последовательных вхождений элемента `6` следующие за ними элементы `5`, `7` и `5` отличаются на единицу от элемента `6`, поэтому они удаляются. Однако следующие вхождения элемента `4` остаются в последовательности, потому что разность между ними и элементом `6` не равна `1`.

Следующий пример сжимает последовательность пробелов:

```
// algo/unique3.cpp

#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

bool bothSpaces (char elem1, char elem2)
{
    return elem1 == ' ' && elem2 == ' ';
}

int main()
{
    // по умолчанию  ведущие пробелы не пропускаются
    cin.unsetf(ios::skipws);

    // копируем стандартный ввод в стандартный вывод
    // - сжимая пробелы
    unique_copy(istream_iterator<char>(cin),  // начало источника: cin
                istream_iterator<char>(),     // конец источника: конец файла
                ostream_iterator<char>(cout), // получатель: cout
                bothSpaces);                  // критерий дубликатов
}
```

Если ввести строку

```
Hello, here are  sometimes more   and  sometimes fewer    spaces.
```

то эта программа выведет следующий результат:

```
Hello, here are sometimes more and sometimes fewer spaces.
```

11.8. Перестановочные алгоритмы

Перестановочные алгоритмы изменяют порядок элементов, но не их значения. Поскольку порядок следования элементов ассоциативных или неупорядоченных контейнеров определяется самим контейнером, эти алгоритмы не могут использовать их в качестве целевых.

11.8.1. Перестановка элементов в обратном порядке

```
void
reverse (BidirectionalIterator beg, BidirectionalIterator end)

OutputIterator
reverse_copy (BidirectionalIterator sourceBeg, BidirectionalIterator
sourceEnd,
              OutputIterator destBeg)
```

- Алгоритм `reverse()` переставляет элементы диапазона [*beg*,*end*) в обратном порядке.
- Алгоритм `reverse_copy()` переставляет элементы в обратном порядке, копируя их из диапазона-источника [*sourceBeg*,*sourceEnd*) в диапазон-получатель, начинающийся с позиции *destBeg*.
- Алгоритм `reverse_copy()` возвращает позицию, находящуюся за последним скопированным элементом в диапазоне-получателе (первый элемент, который не был заменен).
- Вызывающая сторона должна гарантировать, что диапазон-получатель является достаточно большим или что используются итераторы вставки.
- Списки имеют эквивалентную функцию-член `reverse()`, которая обеспечивает более высокую эффективность, потому что она переназначает указатели, а не присваивает значения элементов (см. раздел 8.8.1).
- Сложность: линейная (*numElems*/2 обменов или *numElems* присваиваний соответственно).

Следующая программа демонстрирует использование алгоритмов `reverse()` и `reverse_copy()`:

```
// algo/reverse1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
```

```
{
   vector<int> coll;

   INSERT_ELEMENTS(coll,1,9);
   PRINT_ELEMENTS(coll,"coll: ");

   // переставляем элементы в обратном порядке
   reverse (coll.begin(), coll.end());
   PRINT_ELEMENTS(coll,"coll: ");

   // переставляем элементы в обратном порядке от второго до предпоследнего
   reverse (coll.begin()+1, coll.end()-1);
   PRINT_ELEMENTS(coll,"coll: ");

   // выводим все элементы в обратном порядке
   reverse_copy (coll.cbegin(), coll.cend(),          // источник
                 ostream_iterator<int>(cout," "));    // получатель
   cout << endl;
}
```

Программа выводит следующий результат:

```
coll: 1 2 3 4 5 6 7 8 9
coll: 9 8 7 6 5 4 3 2 1
coll: 9 2 3 4 5 6 7 8 1
1 8 7 6 5 4 3 2 9
```

11.8.2. Циклическая перестановка элементов

Циклическая перестановка элементов в последовательности

`ForwardIterator`
`rotate` (`ForwardIterator` *beg*, `ForwardIterator` *newBeg*, `ForwardIterator` *end*)

- Выполняет циклическую перестановку элементов в диапазоне [*beg*,*end*) так, что `*newBeg` — новый первый элемент после вызова.
- После появления стандарта C++11 возвращает позицию *beg*+(*end*–*newbeg*), представляющую собой новую позицию первого элемента. До появления стандарта C++11 возвращаемое значение имело тип `void`.
- Вызывающая сторона должна гарантировать, что позиция *newBeg* является корректной позицией в диапазоне [*beg*,*end*); в противном случае вызов порождает непредсказуемые последствия.
- Сложность: линейная (не более *numElems* перестановок).

Следующая программа демонстрирует использование алгоритма `rotate()`:

```
// algo/rotate1.cpp

#include "algostuff.hpp"
using namespace std;
```

```
int main()
{
   vector<int> coll;

   INSERT_ELEMENTS(coll,1,9);
   PRINT_ELEMENTS(coll,"coll:      ");

   // выполняем циклический сдвиг на один элемент влево
   rotate (coll.begin(),              // начало диапазона
           coll.begin() + 1,          // новый первый элемент
           coll.end());               // конец диапазона
   PRINT_ELEMENTS(coll,"one left:  ");

   // выполняем циклический сдвиг на два элемента вправо
   rotate (coll.begin(),              // начало диапазона
           coll.end() - 2,            // новый первый элемент
           coll.end());               // конец диапазона
   PRINT_ELEMENTS(coll,"two right: ");

   // выполняем циклический сдвиг так, чтобы элемент со значением 4 стал первым
   rotate (coll.begin(),                    // начало диапазона
           find(coll.begin(),coll.end(),4), // новый первый элемент
           coll.end());                     // конец диапазона
   PRINT_ELEMENTS(coll,"4 first:   ");
}
```

Как показывает этот пример, можно выполнять циклический сдвиг влево с положительным смещением от начала, а сдвиг вправо — с отрицательным смещением от конца. Однако добавление смещения к итератору возможно только для итераторов произвольного доступа (которые поддерживаются векторами). Без таких итераторов следует использовать алгоритм `advance()` (см. пример использования алгоритма `rotate_copy()`).

Программа выводит следующий результат:

```
coll:      1 2 3 4 5 6 7 8 9
one left:  2 3 4 5 6 7 8 9 1
two right: 9 1 2 3 4 5 6 7 8
4 first:   4 5 6 7 8 9 1 2 3
```

Циклический сдвиг элементов при копировании

```
OutputIterator
rotate_copy (ForwardIterator sourceBeg, ForwardIterator newBeg,
             ForwardIterator sourceEnd,
             OutputIterator destBeg)
```

- Представляет собой комбинацию алгоритмов `copy()` и `rotate()`.
- Копирует элементы диапазона-источника [*sourceBeg*,*sourceEnd*) в диапазон-получатель, начинающийся с позиции *destBeg*, с циклическим сдвигом, так, что **newBeg* становится новым первым элементом.
- Возвращает позицию *destBeg*+ (*sourceEnd*-*sourceBeg*), представляющую собой позицию, расположенную за последним скопированным элементом диапазона-получателя.

- Вызывающая сторона должна гарантировать, что позиция *newBeg* является корректной позицией в диапазоне [*beg*,*end*); в противном случае вызов порождает непредсказуемые последствия.
- Вызывающая сторона должна гарантировать, что диапазон-получатель является достаточно большим, или использовать итераторы вставки.
- Источник и получатель не должны перекрываться.
- Сложность: линейная (не более *numElems* присваиваний).

Следующая программа демонстрирует использование алгоритма `rotate_copy()`:

```
// algo/rotate2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   set<int> coll;

   INSERT_ELEMENTS(coll,1,9);
   PRINT_ELEMENTS(coll);

   // выводим элементы, полученные циклическим сдвигом на один элемент влево
   set<int>::const_iterator pos = next(coll.cbegin());
   rotate_copy(coll.cbegin(),                      // начало источника
               pos,                                // новый первый элемент
               coll.cend(),                        // конец источника
               ostream_iterator<int>(cout," "));   // получатель
   cout << endl;

   // выводим элементы, полученные циклическим сдвигом на два элемента вправо
   pos = coll.cend();
   advance(pos,-2);
   rotate_copy(coll.cbegin(),                      // начало источника
               pos,                                // новый первый элемент
               coll.cend(),                        // конец источника
               ostream_iterator<int>(cout," "));   // получатель
   cout << endl;

   // выводим элементы, полученные циклическим сдвигом, так, что
   // элемент со значением 4 оказался первым
   rotate_copy(coll.cbegin(),                      // начало источника
               coll.find(4),                       // новый первый элемент
               coll.cend(),                        // конец источника
               ostream_iterator<int>(cout," "));   // получатель
   cout << endl;
}
```

В отличие от предыдущего примера использования алгоритма `rotate()` (см. раздел 11.8.2), здесь используется множество, а не вектор. Это приводит к двум последствиям.

1. Для того чтобы изменить значение итератора, необходимо использовать алгоритмы `advance()` (см. раздел 9.3.1) или `next()` (см. раздел 9.3.2), потому что двунаправленные итераторы не имеют оператора +.
2. Следует использовать функцию-член `find()`, а не алгоритм `find()`, потому что функция-член обеспечивает более высокую эффективность.

Программа выводит следующий результат:

```
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 1
8 9 1 2 3 4 5 6 7
4 5 6 7 8 9 1 2 3
```

11.8.3. Перестановка элементов

```
bool
next_permutation (BidirectionalIterator beg, BidirectionalIterator end)

bool
next_permutation (BidirectionalIterator beg, BidirectionalIterator end,
                  BinaryPredicate op)

bool
prev_permutation (BidirectionalIterator beg, BidirectionalIterator end)

bool
prev_permutation (BidirectionalIterator beg, BidirectionalIterator end,
                  BinaryPredicate op)
```

- Алгоритм `next_permutation()` изменяет порядок следования элементов диапазона [*beg*,*end*) в соответствии со следующей перестановкой в лексикографическом порядке.
- Алгоритм `prev_permutation()` изменяет порядок следования элементов диапазона [*beg*,*end*) в соответствии с предыдущей перестановкой в лексикографическом порядке.
- Первые формы сравнивают элементы с помощью оператора <.
- Вторые формы сравнивают элементы с помощью бинарного предиката *op*(*elem1*,*elem2*), который должен возвращать значение `true`, если элемент *elem1* "меньше, чем" элемент *elem2*.
- Оба алгоритма возвращают значение `false`, ели элементы образуют "нормальный" (лексикографический) порядок, т.е. возрастающий порядок для алгоритма `next_permutation()` и убывающий порядок для алгоритма `prev_permutation()`.
- Для того чтобы перебрать все перестановки, необходимо упорядочить все элементы (по возрастанию или убыванию) и начать цикл, вызывающий алгоритм `next_permutation()` и `prev_permutation()`, пока эти алгоритмы возвращают значение `true`.[3] Объяснение лексикографической сортировки см. в разделе 11.5.4.
- Сложность: линейная (не более *numElems*/2 обменов).

[3] Алгоритмы `next_permutation()` и `prev_permutation()` можно также использовать для упорядочения элементов в диапазоне. Достаточно просто применять их к диапазону, пока они возвращают значение `true`. Однако этот способ на самом деле очень неэффективный.

Следующий пример демонстрирует перебор всех перестановок с помощью алгоритмов `next_permutation()` и `prev_permutation()`:

```
// algo/permutation1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll;
   INSERT_ELEMENTS(coll,1,3);
   PRINT_ELEMENTS(coll,"on entry: ");

   // переставляем элементы, пока они не будут упорядочены
   // - перебираем все перестановки, поскольку изначально
   //  все элементы упорядочены
   while (next_permutation(coll.begin(),coll.end())) {
          PRINT_ELEMENTS(coll," ");
   }
   PRINT_ELEMENTS(coll,"afterward: ");

   // переставляем, пока элементы не будут упорядочены по возрастанию
   // - это соответствует следующей перестановке после сортировки по убыванию
   // - поэтому цикл останавливается немедленно
   while (prev_permutation(coll.begin(),coll.end())) {
          PRINT_ELEMENTS(coll," ");
   }
   PRINT_ELEMENTS(coll,"now: ");

   // переставляем, пока элементы не будут упорядочены по убыванию
   // - перебираем все перестановки, поскольку изначально
   //  все элементы упорядочены по убыванию
   while (prev_permutation(coll.begin(),coll.end())) {
          PRINT_ELEMENTS(coll," ");
   }
   PRINT_ELEMENTS(coll,"afterward: ");
}
```

Программы выводит следующий результат:

```
on entry: 1 2 3
 1 3 2
 2 1 3
 2 3 1
 3 1 2
 3 2 1
afterward: 1 2 3
now:       3 2 1
 3 1 2
 2 3 1
 2 1 3
 1 3 2
 1 2 3
afterward: 3 2 1
```

11.8.4. Перетасовка элементов

Перетасовка с помощью библиотеки случайных чисел

```
void
shuffle (RandomAccessIterator beg, RandomAccessIterator end,
         UniformRandomNumberGenerator&& eng)

void
random_shuffle (RandomAccessIterator beg, RandomAccessIterator end)

void
random_shuffle (RandomAccessIterator beg, RandomAccessIterator end,
                RandomFunc&& op)
```

- Первая форма, появившаяся вместе со стандартом C++11, случайным образом изменяет порядок следования элементов диапазона [*beg*,*end*), используя библиотечный генератор случайных чисел и распределений (см. раздел 17.1.2).
- Вторая форма случайным образом изменяет порядок следования элементов диапазона [*beg*,*end*), используя зависящий от реализации генератор равномерно распределенных случайных чисел, например, функцию `rand()` из языка C.
- Третья форма случайным образом изменяет порядок следования элементов диапазона [*beg*,*end*), используя операцию *op*, которая вызывается с целочисленным значением типа `difference_type`, определенного в итераторе. Операция *op*(*max*) должна возвращать случайное число, большее или равное нулю, но меньшее *max*. Таким образом, само значение *max* не возвращается.
- Алгоритму `shuffle()` нельзя передавать временный генератор случайных чисел. Подробности изложены в разделе 17.1.1.
- До появления стандарта C++11 операция *op* объявлялась как ссылка `RandomFunc&`, поэтому было невозможно передать временное значение или обычную функцию.
- Сложность: линейная (*numElems*`-1` обменов).

Отметим, что старые глобальные функции языка C, такие как `rand()`, хранили свои локальные состояния в статической переменной. Однако этот способ имеет недостатки: например, генератор случайных чисел по своей природе является небезопасным в смысле потоков, и невозможно получить два независимых потока случайных чисел. Следовательно, лучше использовать функциональные объекты, инкапсулирующие свои локальные состояния в одном или нескольких членах класса. По этой причине при генерировании нового случайного числа алгоритм изменяет состояние генератора, передаваемого как аргумент.

Следующий пример демонстрирует случайную перестановку элементов с помощью алгоритма `random_shuffle()` без передачи генератора случайных чисел и с использованием алгоритма `shuffle()`:

```
// algo/shuffle1.cpp

#include <cstdlib>
#include "algostuff.hpp"
using namespace std;
```

```
int main()
{
   vector<int> coll;
   INSERT_ELEMENTS(coll,1,9);
   PRINT_ELEMENTS(coll,"coll:     ");

   // перетасовываем все элементы
   random_shuffle (coll.begin(), coll.end());
   PRINT_ELEMENTS(coll,"shuffled: ");

   // упорядочиваем их снова
   sort (coll.begin(), coll.end());
   PRINT_ELEMENTS(coll,"sorted:   ");

   // перетасовываем все элементы с помощью генератора, заданного по умолчанию
   default_random_engine dre;
   shuffle (coll.begin(), coll.end(),  // диапазон
            dre);                      // генератор случайных чисел
   PRINT_ELEMENTS(coll,"shuffled: ");
}
```

Возможный (зависящий от платформы) вывод программы:

```
coll:     1 2 3 4 5 6 7 8 9
shuffled: 8 2 4 9 5 7 3 6 1
sorted:   1 2 3 4 5 6 7 8 9
shuffled: 8 7 5 6 2 4 9 3 1
```

Описание генераторов случайных чисел, которые можно передавать алгоритму `shuffle()`, приведено в разделе 17.1.

Следующий пример демонстрирует перетасовку элементов с помощью собственного генератора случайных чисел, передаваемого алгоритму `random_shuffle()` как аргумент:

```
// algo/randomshuffle1.cpp

#include <cstdlib>
#include "algostuff.hpp"
using namespace std;
class MyRandom {
public:
   ptrdiff_t operator() (ptrdiff_t max) {
       double tmp;
       tmp = static_cast<double>(rand())
             / static_cast<double>(RAND_MAX);
       return static_cast<ptrdiff_t>(tmp * max);
   }
};

int main()
{
   vector<int> coll;
   INSERT_ELEMENTS(coll,1,9);
   PRINT_ELEMENTS(coll,"coll:     ");
```

```
    // перетасовываем элементы с помощью собственного генератора случайных чисел
    MyRandom rd;
    random_shuffle (coll.begin(), coll.end(),     // диапазон
                    rd);                          // генератор случайных чисел
    PRINT_ELEMENTS(coll,"shuffled: ");
}
```

При вызове алгоритма `random()` используется собственный генератор случайных чисел `rd()`, вспомогательный функциональный объект класса `MyRandom`, использующий алгоритм генерирования случайных чисел, который часто лучше, чем обычный непосредственный вызов функции `rand()`.[4] Отметим, что до появления стандарта C++11 невозможно было передавать временный объект в качестве генератора случайных чисел так, как показано ниже.

```
random_shuffle (coll.begin(), coll.end(),
                MyRandom());              // ОШИБКА до появления стандарта C++11
```

Возможный (зависящий от платформы) вывод программы:

```
coll:     1 2 3 4 5 6 7 8 9
shuffled: 1 8 6 2 4 9 3 7 5
```

Замечания по поводу использования функции `rand()` приведены в разделе 17.1.1.

11.8.5. Перемещение элементов в начало

```
ForwardIterator
partition (ForwardIterator beg, ForwardIterator end,
           UnaryPredicate op)

BidirectionalIterator
stable_partition (BidirectionalIterator beg, BidirectionalIterator end,
                  UnaryPredicate op)
```

- Оба алгоритма перемещают в начало все элементы диапазона [*beg*,*end*), для которых унарный предикат *op*(*elem*) возвращает значение `true`.
- Оба алгоритма возвращают первую позицию, для которой предикат *op*() возвращает `false`.
- Разница между алгоритмами `partition()` и `stable_partition()` заключается в том, что алгоритм `stable_partition()` сохраняет относительный порядок следования элементов, соответствующих и не соответствующих критерию.
- Этот алгоритм можно использовать для разделения элементов на две части по критерию сортировки. Алгоритм `nth_element()` имеет такую же возможность. Обсуждение разницы между этими алгоритмами и алгоритмом `nth_element()` см. в разделе 11.2.2.
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).

[4] Способ, которым механизм `MyRandom` генерирует случайные числа, описан в книге [*Stroustrup:C++*].

- До появления стандарта C++11 алгоритм `partition()` требовал использования двунаправленных итераторов вместо однонаправленных и гарантировал выполнение не более *numElems*/2 обменов.
- Для копирования элементов в диапазон для элементов, удовлетворяющих предикату, и в диапазон для элементов, не удовлетворяющих критерию, следует использовать алгоритм `partition_copy()` (см. раздел 11.8.6) (доступен начиная со стандарта C++11).
- Сложность:
  - для алгоритма `partition()`: линейная (не более *numElems*/2 обменов и *numElems* вызовов предиката *op*(), если используются двунаправленные итераторы или итераторы произвольного доступа; не более *numElems* обменов, если используются только однонаправленные итераторы);
  - для алгоритма `stable_partition()`: линейная, если есть достаточное количество дополнительной памяти (*numElems* обменов и вызовов предиката *op*()); в противном случае *n*log(n)* (*numElems* вызовов предиката *op*(), но *numElems*log*(*numElems*) обменов).

Следующая программа демонстрирует использование алгоритмов `partition()` и `stable_partition()`, а также разницу между ними:

```
// algo/partition1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll1;
   vector<int> coll2;

   INSERT_ELEMENTS(coll1,1,9);
   INSERT_ELEMENTS(coll2,1,9);
   PRINT_ELEMENTS(coll1,"coll1: ");
   PRINT_ELEMENTS(coll2,"coll2: ");
   cout << endl;

   // перемещаем все четные элементы в начало
   vector<int>::iterator pos1, pos2;
   pos1 = partition(coll1.begin(), coll1.end(),          // диапазон
                    [](int elem){                        // критерий
                       return elem%2==0;
                    });
   pos2 = stable_partition(coll2.begin(), coll2.end(),   // диапазон
                           [](int elem){                 // критерий
                              return elem%2==0;
                           });

   // выводим коллекции и первый нечетный элемент
   PRINT_ELEMENTS(coll1,"coll1: ");
   cout << "first odd element: " << *pos1 << endl;
```

```
   PRINT_ELEMENTS(coll2,"coll2: ");
   cout << "first odd element: " << *pos2 << endl;
}
```

Программа выводит следующий результат:

```
coll1: 1 2 3 4 5 6 7 8 9
coll2: 1 2 3 4 5 6 7 8 9
coll1: 8 2 6 4 5 3 7 1 9
first odd element: 5
coll2: 2 4 6 8 1 3 5 7 9
first odd element: 1
```

Как показывает этот пример, алгоритм `stable_partition()`, в отличие от алгоритма `partition()`, сохраняет относительный порядок следования четных и нечетных элементов.

11.8.6. Разделение на два подынтервала

```
pair<OutputIterator1,OutputIterator2>
partition_copy (InputIterator sourceBeg, InputIterator sourceEnd,
                OutputIterator1 destTrueBeg, OutputIterator2 destFalseBeg,
                UnaryPredicate op)
```

- Разделяет все элементы диапазона [*beg*,*end*) на два подынтервала в соответствии с предикатом *op*().
- Все элементы, для которых унарный предикат *op*(*elem*) возвращает значение `true`, копируются в диапазон, начинающийся с позиции *destTrueBeg*. Все элементы, для которых предикат возвращает значение `false`, копируются в диапазон, начинающийся с позиции *destFalseBeg*.
- Алгоритм возвращает пару позиций, находящихся после последних скопированных элементов в целевых диапазонах (позиции первых не перезаписанных элементов).
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова функции (см. раздел 10.1.4).
- Этот алгоритм доступен начиная со стандарта C++11.
- Если нужны только элементы, удовлетворяющие предикату, или элементы, не удовлетворяющие предикату, следует использовать алгоритм `copy_if()` (см. раздел 11.6.1) или `remove_copy_if()` (см. раздел 11.7.1).
- Сложность: линейная (не более *numElems* вызовов предиката *op*()).

Следующая программа демонстрирует алгоритм `partition_copy()`:

```
// algo/partitioncopy1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll = { 1, 6, 33, 7, 22, 4, 11, 33, 2, 7, 0, 42, 5 };
   PRINT_ELEMENTS(coll,"coll:     ");
```

```
    // коллекции-получатели:
    vector<int> evenColl;
    vector<int> oddColl;

    // копируем все элементы, соответствующим образом разделенные
    // на четные и нечетные элементы
    partition_copy (coll.cbegin(),coll.cend(), // диапазон-источник
                    back_inserter(evenColl),   // диапазон для четных элементов
                    back_inserter(oddColl),    // диапазон для нечетных элементов
                    [](int elem){              // предикат: проверка четности
                       return elem%2==0;
                    });
    PRINT_ELEMENTS(evenColl,"evenColl: ");
    PRINT_ELEMENTS(oddColl, "oddColl:  ");
}
```

Программа выводит следующий результат:

```
coll:     1 6 33 7 22 4 11 33 2 7 0 42 5
evenColl: 6 22 4 2 0 42
oddColl:  1 33 7 11 33 7 5
```

11.9. Алгоритмы сортировки

В библиотеке STL есть несколько алгоритмов для сортировки элементов диапазона. Кроме полной сортировки, в библиотеке STL предусмотрены варианты частичной сортировки. Если задача позволяет, следует отдавать предпочтение алгоритмам частичной сортировки, поскольку обычно они имеют более высокую эффективность.

Поскольку (последовательные) списки, а также ассоциативные и неупорядоченные контейнеры не имеют итераторов произвольного доступа, эти контейнеры нельзя использовать в качестве диапазонов-получателей алгоритмов сортировки. Вместо этого ассоциативные контейнеры можно использовать для автоматической сортировки. Тем не менее однократная сортировка обычно оказывается эффективнее, чем постоянное поддержание порядка в контейнере (см. раздел 7.12).

11.9.1. Сортировка всех элементов

```
void
sort (RandomAccessIterator beg, RandomAccessIterator end)

void
sort (RandomAccessIterator beg, RandomAccessIterator end,
      BinaryPredicate op)

void
stable_sort (RandomAccessIterator beg, RandomAccessIterator end)

void
stable_sort (RandomAccessIterator beg, RandomAccessIterator end,
             BinaryPredicate op)
```

- Первые формы алгоритмов `sort()` и `stable_sort()` упорядочивают все элементы диапазона [*beg*,*end*) с помощью оператора <.
- Вторые формы алгоритмов `sort()` и `stable_sort()` упорядочивают все элементы с помощью бинарного предиката *op*(*elem1*,*elem2*), который играет роль критерия сортировки. Он должен возвращать значение `true`, если элемент *elem1* "меньше, чем" элемент *elem2*.
- Отметим, что предикат *op* должен определять *строгое слабое упорядочение* значений (см. раздел 7.7).
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова (см. раздел 10.1.4).
- Разница между алгоритмами `sort()` и `stable_sort()` заключается в том, что второй из них гарантирует, что порядок одинаковых элементов останется неизменным.
- Эти алгоритмы нельзя применять к обычным и последовательным спискам, поскольку оба эти контейнера не имеют итераторов произвольного доступа. Однако эти контейнеры предусматривают специальную функцию-член `sort()`, предназначенную для сортировки (см. раздел 8.8.1).
- Алгоритм `sort()` гарантирует высокую эффективность (*n*log(n)*). Однако до появления стандарта C++11 он гарантировал эту эффективность только в среднем. Таким образом, если важно избежать худшего варианта, следует использовать алгоритмы `partial_sort()` и `stable_sort()`. Обсуждение алгоритмов сортировки см. в разделе 11.2.2.
- Сложность:
  - для алгоритма `sort()`: *n*log(n)* (примерно *numElems**`log(`*numElems*`)` сравнений в среднем); до появления стандарта C++11 сложность *n**log*n* гарантировалась только в среднем;
  - для алгоритма `stable_sort()`: *n*log(n)*, если есть достаточный объем дополнительной памяти (*numElems***log* (*numElems*) сравнений); в противном случае *n*log(n)*log (n)**(*numElems** `(log(`*numElems*`))`^{2} сравнений).

Следующий пример демонстрирует использование алгоритма `sort()`:

```
// algo/sort1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   deque<int> coll;

   INSERT_ELEMENTS(coll,1,9);
   INSERT_ELEMENTS(coll,1,9);

   PRINT_ELEMENTS(coll,"on entry: ");

   // сортируем элементы
```

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

   PRINT_ELEMENTS(coll,"sorted:   ");

   // упорядочение в обратном порядке
   sort (coll.begin(), coll.end(),        // диапазон
         greater<int>());                 // критерий сортировки
   PRINT_ELEMENTS(coll,"sorted >: ");
}
```

Программа выводит следующий результат:

```
on entry: 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
sorted:   1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
sorted >: 9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1
```

Пример, демонстрирующий сортировку элементов с помощью функции-члена класса, см. в разделе 6.8.2.

Следующая программа демонстрирует различия между алгоритмами `sort()` и `stable_sort()`. В ней оба алгоритма используются для сортировки строк по количеству их символов на основе критерия сортировки `lessLength()`:

```
// algo/sort2.cpp

#include "algostuff.hpp"
using namespace std;

bool lessLength (const string& s1, const string& s2)
{
   return s1.length() < s2.length();
}

int main()
{
   // заполняем две коллекции одинаковыми элементами
   vector<string> coll1 = { "1xxx", "2x", "3x", "4x", "5xx", "6xxxx",
                            "7xx", "8xxx", "9xx", "10xxx", "11", "12",
                            "13", "14xx", "15", "16", "17" };
   vector<string> coll2(coll1);

   PRINT_ELEMENTS(coll1,"on entry:\n ");

   // сортируем (по длине строк)
   sort (coll1.begin(), coll1.end(),          // диапазон
         lessLength);                         // критерий
   stable_sort (coll2.begin(), coll2.end(),   // диапазон
                lessLength);                  // критерий

   PRINT_ELEMENTS(coll1,"\nwith sort():\n ");
   PRINT_ELEMENTS(coll2,"\nwith stable_sort():\n ");
}
```

Программа выводит следующий результат:

```
on entry:
1xxx 2x 3x 4x 5xx 6xxxx 7xx 8xxx 9xx 10xxx 11 12 13 14xx 15 16 17
with sort():
2x 3x 4x 17 16 15 13 12 11 9xx 7xx 5xx 1xxx 8xxx 14xx 10xxx 6xxxx
with stable_sort():
2x 3x 4x 11 12 13 15 16 17 5xx 7xx 9xx 1xxx 8xxx 14xx 6xxxx 10xxx
```

Только алгоритм `stable_sort()` сохраняет относительный порядок элементов (который определяется начальными числами коллекции).

11.9.2. Частичная сортировка

```
void
partial_sort (RandomAccessIterator beg, RandomAccessIterator sortEnd,
              RandomAccessIterator end)

void
partial_sort (RandomAccessIterator beg, RandomAccessIterator sortEnd,
              RandomAccessIterator end, BinaryPredicate op)
```

- Первая форма сортирует элементы диапазона [*beg*,*end*) с помощью оператора <, так что диапазон [*beg*,*sortEnd*) содержит упорядоченные элементы.
- Вторая форма сортирует элементы с помощью бинарного предиката *op* (*elem1*, *elem2*), играющего роль критерия сортировки, так что диапазон [*beg*,*sortEnd*) содержит упорядоченные элементы.
- Отметим, что предикат *op* должен определять *строгое слабое упорядочение* значений (см. раздел 7.7).
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова (см. раздел 10.1.4).
- В отличие от алгоритма `sort()`, алгоритм `partial_sort()` не сортирует все элементы, а останавливается после упорядочения элементов от первой позиции до позиции *sortEnd*. Таким образом, если после сортировки последовательности вам потребуются только первые три элемента, этот алгоритм сэкономит время, потому что не будет выполнять ненужную сортировку остальных элементов.
- Если позиция *sortEnd* совпадает с позицией *end*, то алгоритм `partial_sort()` сортирует всю последовательность. В среднем его эффективность ниже, чем у алгоритма `sort()`, но в худшем случае — выше. Обсуждение алгоритмов сортировки см. в разделе 11.2.2.
- Сложность: между линейной и *n*log(n)* (примерно *numElems* log* (*numSortedElems*) сравнений).

Следующая программа демонстрирует использование алгоритма `partial_sort()`:

```
// algo/partialsort1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
```

```
    deque<int> coll;

    INSERT_ELEMENTS(coll,3,7);
    INSERT_ELEMENTS(coll,2,6);
    INSERT_ELEMENTS(coll,1,5);

    PRINT_ELEMENTS(coll);

    // упорядочиваем первые пять элементов
    partial_sort (coll.begin(),              // начало диапазона
                  coll.begin()+5,            // конец упорядоченного диапазона
                  coll.end());               // конец всего диапазона
    PRINT_ELEMENTS(coll);

    // обратная сортировка первых пяти элементов
    partial_sort (coll.begin(),              // начало диапазона
                  coll.begin()+5,            // конец упорядоченного диапазона
                  coll.end(),                // конец всего диапазона
                  greater<int>());           // критерий сортировки
    PRINT_ELEMENTS(coll);

    // сортируем все элементы
    partial_sort (coll.begin(),              // начало диапазона
                  coll.end(),                // конец упорядоченного диапазона
                  coll.end());               // конец всего диапазона
    PRINT_ELEMENTS(coll);
}
```

Программа выводит следующий результат:

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

```
RandomAccessIterator
partial_sort_copy (InputIterator sourceBeg, InputIterator sourceEnd,
                   RandomAccessIterator destBeg, RandomAccessIterator destEnd)

RandomAccessIterator
partial_sort_copy (InputIterator sourceBeg, InputIterator sourceEnd,
                   RandomAccessIterator destBeg, RandomAccessIterator destEnd,
                   BinaryPredicate op)
```

- Обе формы представляют собой комбинацию алгоритмов `copy()` и `partial_sort()`.
- Обе формы копируют упорядоченные элементы из диапазона-источника [*sourceBeg*,*sourceEnd*) в диапазон-получатель [*destBeg*,*destEnd*).
- Количество копируемых и упорядочиваемых элементов равно минимальному количеству элементов в диапазоне-источнике и диапазоне-получателе.

- Обе формы возвращают позицию, расположенную за последним скопированным элементом в диапазоне-получателе (позиция первого элемента, который не был заменен).
- Если размер диапазона-источника [*sourceBeg*,*sourceEnd*) не больше, чем размер диапазона-получателя [*destBeg*,*destEnd*), то копируются и упорядочиваются все элементы. Таким образом, алгоритм работает как комбинация алгоритмов `copy()` и `sort()`.
- Отметим, что предикат *op* должен определять *строгое слабое упорядочение* значений (см. раздел 7.7).
- Сложность: между линейной и *n*logn* (примерно *numElems*log*(*numSortedElems*) сравнений).

Следующая программа демонстрирует несколько примеров использования алгоритма `partial_sort_copy()`:

```
// algo/partialsort2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   deque<int> coll1;
   vector<int> coll6(6);          // инициализируем 6 элементами
   vector<int> coll30(30);        // инициализируем 30 элементами

   INSERT_ELEMENTS(coll1,3,7);
   INSERT_ELEMENTS(coll1,2,6);
   INSERT_ELEMENTS(coll1,1,5);

   PRINT_ELEMENTS(coll1);

   // копируем и упорядочиваем элементы из coll1 в coll6
   vector<int>::const_iterator pos6;
   pos6 = partial_sort_copy (coll1.cbegin(), coll1.cend(),
                             coll6.begin(), coll6.end());

   // выводим все скопированные элементы
   copy (coll6.cbegin(), pos6,
         ostream_iterator<int>(cout," "));
   cout << endl;

   // копируем и упорядочиваем элементы из coll1 в coll30
   vector<int>::const_iterator pos30;
   pos30 = partial_sort_copy (coll1.cbegin(), coll1.cend(),
                              coll30.begin(), coll30.end(),
                              greater<int>());
   // выводим все скопированные элементы
   copy (coll30.cbegin(), pos30,
         ostream_iterator<int>(cout," "));
```

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

Программа выводит следующий результат:

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

Диапазон-получатель в первом вызове алгоритма `partial_sort_copy()` содержит только шесть элементов и возвращает конец коллекции `coll6`. Второй вызов алгоритма `partial_sort_copy()` копирует все элементы коллекции `coll1` в коллекцию `coll30`, имеющую достаточный размер, так что все элементы копируются и упорядочиваются.

11.9.3. Сортировка по *n*-му элементу

```
void
nth_element (RandomAccessIterator beg, RandomAccessIterator nth,
             RandomAccessIterator end)

void
nth_element (RandomAccessIterator beg, RandomAccessIterator nth,
             RandomAccessIterator end, BinaryPredicate op)
```

- Обе формы сортируют элементы диапазона [*beg*,*end*) так, чтобы правильный элемент находился в *n*-й позиции, а все элементы, стоящие перед ним, не превосходили его по величине или были равны ему, и все элементы, стоящие после него, превосходили его по величине или были равны ему. Таким образом, получаются две последовательности, разделенные элементом, стоящим в позиции *n*, причем любой элемент первой подпоследовательности меньше или равен элементу второй подпоследовательности. Это удобно, если нужно определить первые *n* наименьших или наибольших элементов, не сортируя их.
- Первая форма использует в качестве критерия сортировки оператор <.
- Вторая форма использует в качестве критерия сортировки бинарный предикат *op* (*elem1*, *elem2*).
- Отметим, что предикат *op* должен определять *строгое слабое упорядочение* значений (см. раздел 7.7).
- Отметим, что предикат *op* не должен изменять свое состояние во время вызова (см. раздел 10.1.4).
- Для разделения последовательности на две части по критерию сортировки используется также алгоритм `partition()` (см. раздел 11.8.5). Обсуждение различий между алгоритмами `nth_element()` и `partition()` приведено в разделе 11.2.2.
- Сложность: в среднем линейная.

Следующая программа демонстрирует использование алгоритма `nth_element()`:

```
// algo/nthelement1.cpp

#include "algostuff.hpp"
```

```
using namespace std;

int main()
{
   deque<int> coll;

   INSERT_ELEMENTS(coll,3,7);
   INSERT_ELEMENTS(coll,2,6);
   INSERT_ELEMENTS(coll,1,5);
   PRINT_ELEMENTS(coll);

   // извлекаем четыре наименьших элемента
   nth_element (coll.begin(),                  // начало диапазона
                coll.begin()+3,                // граничный элемент
                coll.end());                   // конец диапазона

   // выводим элементы
   cout << "the four lowest elements are:  ";
   copy (coll.cbegin(), coll.cbegin()+4,
         ostream_iterator<int>(cout," "));
   cout << endl;

   // извлекаем четыре наибольших элемента
   nth_element (coll.begin(),                  // начало диапазона
                coll.end()-4,                  // граничный элемент
                coll.end());                   // конец диапазона

   // выводим элементы
   cout << "the four highest elements are: ";
   copy (coll.cend()-4, coll.cend(),
         ostream_iterator<int>(cout," "));
   cout << endl;

   // извлекаем четыре наибольших элемента (вторая версия)
   nth_element (coll.begin(),                  // начало диапазона
                coll.begin()+3,                // граничный элемент
                coll.end(),                    // конец диапазона
                greater<int>());               // критерий сортировки

   // выводим элементы
   cout << "the four highest elements are: ";
   copy (coll.cbegin(), coll.cbegin()+4,
         ostream_iterator<int>(cout," "));
   cout << endl;
}
```

Программа выводит следующий результат:

```
3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
the four lowest elements are:  2 1 2 3
the four highest elements are: 5 6 7 6
the four highest elements are: 6 7 6 5
```

11.9.4. Алгоритмы для работы с пирамидой

В контексте сортировки *пирамида* представляет собой специальный механизм для сортировки элементов. Она используется в алгоритме *heapsort*. Пирамида — это бинарное дерево, реализованное в виде последовательной коллекции. Пирамиды обладают двумя свойствами[5].

1. Первый элемент всегда является наибольшим (или одним из нескольких наибольших элементов).
2. Добавление или удаление элемента имеет логарифмическую сложность.

Пирамида — это идеальный способ реализации очереди с приоритетами, т.е. очереди, автоматически упорядочивающей свои элементы так, чтобы следующий элемент всегда был наибольшим (или одним из наибольших). По этой причине алгоритмы для работы с пирамидой используются контейнером `priority_queue` (см. раздел 12.3). Библиотека STL предоставляет четыре алгоритма для работы с пирамидой.

1. Алгоритм `make_heap()` преобразовывает диапазон элементов в пирамиду.
2. Алгоритм `push_heap()` добавляет один элемент в пирамиду.
3. Алгоритм `pop_heap()` удаляет следующий элемент из пирамиды.
4. Алгоритм `sort_heap()` преобразовывает пирамиду в упорядоченную коллекцию, после чего она перестает быть пирамидой.

Кроме того, в стандарта C++11 в библиотеку STL включены алгоритмы `is_heap()` и `is_heap_until()` для проверки, является ли коллекция пирамидой, и для возвращения первого элемента, нарушающего свойства пирамиды соответственно (см. раздел 11.5.5).

Как обычно, в качестве критерия сортировки можно передавать бинарный предикат. По умолчанию критерием сортировки является оператор <.

Подробное описание алгоритмов для работы с пирамидой

```
void
make_heap (RandomAccessIterator beg, RandomAccessIterator end)

void
make_heap (RandomAccessIterator beg, RandomAccessIterator end,
           BinaryPredicate op)
```

- Обе формы преобразовывают элементы диапазона [*beg*,*end*) в пирамиду.
- Аргумент *op* является необязательным бинарным предикатом, используемым в качестве критерия сортировки:

  op (*elem1*, *elem2*)
- Эти функции нужны только для начала работы с пирамидой, состоящей из нескольких элементов (один элемент по определению является пирамидой).
- Сложность: линейная (не более 3**numElems* сравнений).

---

[5] На самом деле основное свойство (невозрастающей) пирамиды заключается в том, что значение каждого некорневого узла не превышает значение его родительского узла. Подробно о пирамидах и их реализации можно прочесть в главе 6 книги Кормен Т. и др. *Алгоритмы: построение и анализ, 3-е изд.* — М.: ООО "И.Д. Вильямс", 2013. — *Примеч. консульт.*

```
void
push_heap (RandomAccessIterator beg, RandomAccessIterator end)
void
push_heap (RandomAccessIterator beg, RandomAccessIterator end,
           BinaryPredicate op)
```

- Обе формы добавляют последний элемент, стоящий перед позицией *end*, в существующую пирамиду в диапазоне [*beg*,*end*-1), так что весь диапазон [*beg*,*end*) становится пирамидой.
- Аргумент *op* является необязательным бинарным предикатом, используемым в качестве критерия сортировки:

  op (*elem1*, *elem2*)
- Вызывающая сторона должна гарантировать, что элементы диапазона [*beg*,*end*-1) образуют пирамиду (в соответствии с тем же критерием сортировки), а новый элемент следует непосредственно за этими элементами.
- Сложность: логарифмическая (не более *log* (*numElems*) сравнений).

```
void
pop_heap (RandomAccessIterator beg, RandomAccessIterator end)
void
pop_heap (RandomAccessIterator beg, RandomAccessIterator end,
          BinaryPredicate op)
```

- Обе формы перемещают наибольший элемент пирамиды [*beg*,*end*), являющийся ее первым элементом, в последнюю позицию и создают новую пирамиду из оставшихся элементов диапазона [*beg*,*end*-1).
- Аргумент *op* является необязательным бинарным предикатом, используемым в качестве критерия сортировки:

  op (*elem1*, *elem2*)
- Вызывающая сторона должна гарантировать, что элементы диапазона [*beg*,*end*-1) образуют пирамиду (в соответствии с тем же критерием сортировки).
- Сложность: логарифмическая (не более 2**log* (*numElems*) сравнений).

```
void
sort_heap (RandomAccessIterator beg, RandomAccessIterator end)
void
sort_heap (RandomAccessIterator beg, RandomAccessIterator end,
           BinaryPredicate op)
```

- Обе формы преобразовывают пирамиду [*beg*,*end*) в упорядоченную последовательность.
- Аргумент *op* является необязательным бинарным предикатом, используемым в качестве критерия сортировки:

  op (*elem1*, *elem2*)
- После вызова диапазон перестает быть пирамидой.

- Вызывающая сторона должна гарантировать, что элементы диапазона [*beg*,*end*) образуют пирамиду (в соответствии с тем же критерием сортировки).
- Сложность: *n*log(n)* (не более *numElems*log* (*numElems*) сравнений).

Примеры использования пирамиды

Следующая программа демонстрирует использование разных алгоритмов для работы с пирамидой:

```
// algo/heap1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll;

   INSERT_ELEMENTS(coll,3,7);
   INSERT_ELEMENTS(coll,5,9);
   INSERT_ELEMENTS(coll,1,4);
   PRINT_ELEMENTS (coll, "on entry:          ");

   // преобразовываем коллекцию в пирамиду
   make_heap (coll.begin(), coll.end());

   PRINT_ELEMENTS (coll, "after make_heap(): ");

   // выталкиваем следующий элемент из пирамиды
   pop_heap (coll.begin(), coll.end());
             coll.pop_back();
   PRINT_ELEMENTS (coll, "after pop_heap():  ");

   // заталкиваем новый элемент в пирамиду
   coll.push_back (17);
   push_heap (coll.begin(), coll.end());
   PRINT_ELEMENTS (coll, "after push_heap(): ");

   // преобразовываем пирамиду в упорядоченную коллекцию
   // - ПРИМЕЧАНИЕ: после выполнения вызова она перестанет быть пирамидой
   sort_heap (coll.begin(), coll.end());
   PRINT_ELEMENTS (coll, "after sort_heap(): ");
}
```

Программа выводит следующий результат:

```
on entry:          3 4 5 6 7 5 6 7 8 9 1 2 3 4
after make_heap(): 9 8 6 7 7 5 5 3 6 4 1 2 3 4
after pop_heap():  8 7 6 7 4 5 5 3 6 4 1 2 3
after push_heap(): 17 7 8 7 4 5 6 3 6 4 1 2 3 5
after sort_heap(): 1 2 3 3 4 4 5 5 6 6 7 7 8 17
```

После выполнения алгоритма `make_heap()` элементы упорядочиваются в виде пирамиды в соответствии с ее свойством.

```
9 8 6 7 7 5 5 3 6 4 1 2 3 4
```

Преобразуйте элементы в бинарное дерево, и вы увидите, что значение каждого узла меньше или равно значению родительского узла (рис. 11.1). Алгоритмы `push_heap()` и `pop_heap()` изменяют элементы так, что инвариант бинарного дерева — каждый узел не больше родительского — остается неизменным.

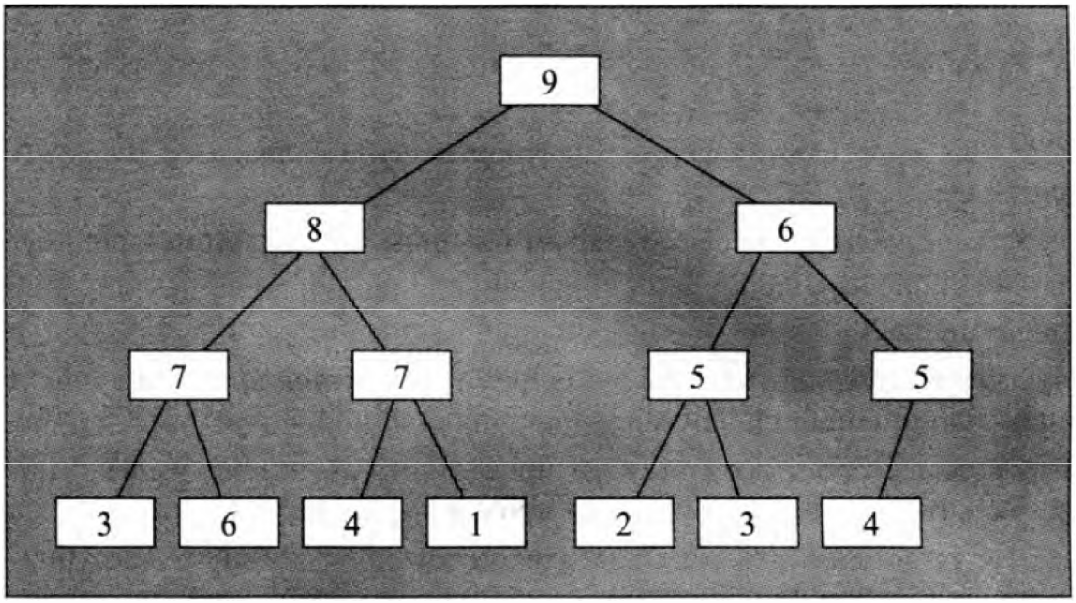


Рис. 11.1. Элементы пирамиды в виде бинарного дерева

11.10. Алгоритмы для упорядоченных диапазонов

Для использования алгоритмов для упорядоченных диапазонов необходимо, чтобы диапазоны-источники содержали элементы, упорядоченные в соответствии с критерием сортировки. Эти алгоритмы могут быть значительно эффективнее, чем аналогичные алгоритмы для неупорядоченных диапазонов (обычно они имеют логарифмическую, а не линейную сложность). Эти алгоритмы можно использовать в сочетании с итераторами, которые не являются итераторами произвольного доступа. Однако в этом случае алгоритмы имеют линейную сложность, потому что они должны перебирать последовательность поэлементно. Тем не менее количество сравнений может быть по-прежнему логарифмическим.

В соответствии со стандартом, вызов этих алгоритмов для неупорядоченных последовательностей приводит к непредсказуемым последствиям. Однако в большинстве реализаций применение этих алгоритмов к неупорядоченным последовательностям допускается. Но все же полагаться на этот факт не рекомендуется, поскольку при этом не обеспечивается переносимость программ.

Ассоциативные и неупорядоченные контейнеры имеют специальные функции-члены для реализации рассматриваемых в этом разделе алгоритмов поиска. При поиске конкретного значения или ключа следует использовать именно их.

11.10.1. Поиск элементов

Следующие алгоритмы выполняют поиск заданных значений в упорядоченных диапазонах.

Проверка наличия элемента

```
bool
binary_search (ForwardIterator beg, ForwardIterator end, const T& value)

bool
binary_search (ForwardIterator beg, ForwardIterator end, const T& value,
               BinaryPredicate op)
```

- Обе формы проверяют, содержит ли упорядоченный диапазон [*beg*,*end*) элемент, равный *value*.
- Аргумент *op* является необязательным бинарным предикатом, используемым как критерий сортировки:

  op (*elem1*, *elem2*)
- Для получения позиции искомого элемента используется алгоритм `lower_bound()`, `upper_bound()` или `equal_range()`.
- Вызывающая сторона должна гарантировать, что диапазоны изначально упорядочены в соответствии с указанным критерием сортировки.
- Сложность: логарифмическая для итераторов произвольного доступа, линейная в противном случае (не более *log* (*numElems*) +2 сравнений; но для итераторов, не являющихся итераторами произвольного доступа, количество операций, необходимых для перебора элементов, является линейным, что в результате порождает общую линейную сложность).

Следующая программа демонстрирует использование алгоритма `binary_search()`:

```
// algo/binarysearch1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   list<int> coll;

   INSERT_ELEMENTS(coll,1,9);
  PRINT_ELEMENTS(coll);

   // проверка существования элемента, равного 5
   if (binary_search(coll.cbegin(), coll.cend(), 5)) {
                    cout << "5 is present" << endl;
   }
   else {
       cout << "5 is not present" << endl;
   }
```

```
    // проверка существования элемента, равного 42
    if (binary_search(coll.cbegin(), coll.cend(), 42)) {
        cout << "42 is present" << endl;
    }
    else {
        cout << "42 is not present" << endl;
    }
}
```

Программа выдает следующий результат:

```
1 2 3 4 5 6 7 8 9
5 is present
42 is not present
```

Проверка наличия нескольких элементов

```
bool
includes (InputIterator1 beg, InputIterator1 end,
          InputIterator2 searchBeg, InputIterator2 searchEnd)

bool
includes (InputIterator1 beg, InputIterator1 end,
          InputIterator2 searchBeg, InputIterator2 searchEnd,
          BinaryPredicate op)
```

- Обе формы проверяют, содержит ли упорядоченный диапазон [*beg*,*end*) все элементы упорядоченного диапазона [*searchBeg*,*searchEnd*), т.е. существует ли для каждого элемента в диапазоне [*searchBeg*,*searchEnd*) равный ему элемент в диапазоне [*beg*,*end*). Если элементы в диапазоне [*searchBeg*,*searchEnd*) равны друг другу, то диапазон [*beg*,*end*) должен содержать такое же количество элементов. Иначе говоря, диапазон [*searchBeg*,*searchEnd*) должен быть подмножеством диапазона [*beg*,*end*).
- Аргумент *op* является необязательным бинарным предикатом, используемым как критерий сортировки:

  op(*elem1*, *elem2*)
- Вызывающая сторона должна гарантировать, что диапазоны изначально упорядочены в соответствии с указанным критерием сортировки.
- Сложность: линейная (не более `2 *` (*numElems*+*numSearchElems*) `- 1` сравнений).

Следующая программа демонстрирует использование алгоритма `includes()`:

```
// algo/includes1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;
    vector<int> search;
```

```
    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll:   ");

    search.push_back(3);
    search.push_back(4);
    search.push_back(7);
    PRINT_ELEMENTS(search,"search: ");

  // проверяем, все ли искомые элементы содержатся в коллекции coll
    if (includes (coll.cbegin(), coll.cend(),
        search.cbegin(), search.cend())) {
    cout << "all elements of search are also in coll"
         << endl;
    }
     else {
        cout << "not all elements of search are also in coll"
             << endl;
    }
}
```

Программа выводит следующий результат:

```
coll:   1 2 3 4 5 6 7 8 9
search: 3 4 7
all elements of search are also in coll
```

Поиск первой или последней возможной позиции

```
ForwardIterator
lower_bound (ForwardIterator beg, ForwardIterator end, const T& value)

ForwardIterator
lower_bound (ForwardIterator beg, ForwardIterator end, const T& value,
             BinaryPredicate op)

ForwardIterator
upper_bound (ForwardIterator beg, ForwardIterator end, const T& value)

ForwardIterator
upper_bound (ForwardIterator beg, ForwardIterator end, const T& value,
             BinaryPredicate op)
```

- Алгоритм `lower_bound()` возвращает позицию первого элемента, который не меньше значения *value*. Это первая позиция, в которую можно вставить элемент со значением *value* без нарушения порядка элементов в диапазоне [*beg*,*end*).
- Алгоритм `upper_bound()` возвращает позицию последнего элемента, превышающего значение *value*. Это последняя позиция, в которую можно вставить элемент со значением *value* без нарушения порядка элементов в диапазоне [*beg*,*end*).
- Если искомого элемента нет, все алгоритмы возвращают позицию *end*.
- Аргумент *op* является необязательным бинарным предикатом, используемым как критерий сортировки:

  op (*elem1*, *elem2*)

- Вызывающая сторона должна гарантировать, что диапазоны изначально упорядочены в соответствием с критерием сортировки.
- Для того чтобы получить результат работы обоих алгоритмов — `lower_bound()` и `upper_bound()`, — используйте алгоритм `equal_range()` (см. описание следующего алгоритма).
- Ассоциативные контейнеры предоставляют эквивалентные функции-члены, обеспечивающие более высокую эффективность (см. раздел 8.3.3).
- Сложность: логарифмическая для итераторов произвольного доступа, линейная в противном случае (не более *log*(*numElems*) +1 сравнений; но для итераторов, не являющихся итераторами произвольного доступа, количество операций, необходимых для перебора элементов, является линейным, что в результате порождает общую линейную сложность).

Следующая программа демонстрирует использование алгоритмов `lower_bound()` и `upper_bound()`:

```
// algo/bounds1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   list<int> coll;

   INSERT_ELEMENTS(coll,1,9);
   INSERT_ELEMENTS(coll,1,9);
   coll.sort ();
   PRINT_ELEMENTS(coll);

   // выводим первую и последнюю позиции, в которые можно вставить 5
   auto pos1 = lower_bound (coll.cbegin(), coll.cend(),
                            5);
   auto pos2 = upper_bound (coll.cbegin(), coll.cend(),
                            5);
   cout << "5 could get position "
        << distance(coll.cbegin(),pos1) + 1
        << " up to "
        << distance(coll.cbegin(),pos2) + 1
        << " without breaking the sorting" << endl;

   // вставляем 3 в первую возможную позицию, не нарушая порядок элементов
   coll.insert (lower_bound(coll.begin(),coll.end(),
                3),
                3);
   // вставляем 7 в последнюю возможную позицию, не нарушая порядок элементов
   coll.insert (upper_bound(coll.begin(),coll.end(),
                7),
                7);
   PRINT_ELEMENTS(coll);
}
```

Программа выводит следующий результат:

```
1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
5 could get position 9 up to 11 without breaking the sorting
1 1 2 2 3 3 3 4 4 5 5 6 6 7 7 7 8 8 9 9
pos1 and pos2 have type
list<int>::const_iterator
```

Поиск первой и последней возможной позиции

```
pair<ForwardIterator,ForwardIterator>
equal_range (ForwardIterator beg, ForwardIterator end, const T& value)

pair<ForwardIterator,ForwardIterator>
equal_range (ForwardIterator beg, ForwardIterator end, const T& value,
             BinaryPredicate op)
```

- Обе формы возвращают диапазон элементов, равных значению *value*. Диапазон содержит первую и последнюю позиции, в которые можно вставить элемент, имеющий значение *value*, не нарушая упорядоченность диапазона [*beg*,*end*).
- Это эквивалентно выражению

  `make_pair(lower_bound(...),upper_bound(...))`
- Аргумент *op* — это необязательный бинарный предикат, используемый в качестве критерия сортировки:

  op(*elem1*, *elem2*)
- Вызывающая сторона должна гарантировать, что диапазоны изначально упорядочены в соответствии с указанным критерием сортировки.
- Ассоциативные контейнеры имеют эквивалентные функции-члены, обеспечивающие более высокую эффективность (см. раздел 8.3.3).
- Сложность: логарифмическая для итераторов произвольного доступа, линейная в противном случае (не более 2**log*(*numElems*) +1 сравнений; но для итераторов, не являющихся итераторами произвольного доступа, количество операций, необходимых для перебора элементов, является линейным, что в результате порождает общую линейную сложность).

Следующая программа демонстрирует использование алгоритма `equal_range()`:

```
// algo/equalrange1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   list<int> coll;

   INSERT_ELEMENTS(coll,1,9);
   INSERT_ELEMENTS(coll,1,9);
   coll.sort ();
```

```
    PRINT_ELEMENTS(coll);

    // выводим первую и последнюю позиции, в которые можно вставить 5
    pair<list<int>::const_iterator,list<int>::const_iterator> range;
    range = equal_range (coll.cbegin(), coll.cend(),
                         5);
    cout << "5 could get position "
         << distance(coll.cbegin(),range.first) + 1
         << " up to "
         << distance(coll.cbegin(),range.second) + 1
         << " without breaking the sorting" << endl;
}
```

Программа выводит следующий результат:

```
1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
5 could get position 9 up to 11 without breaking the sorting
```

11.10.2. Слияние диапазонов

Следующие алгоритмы выполняют слияние элементов двух диапазонов. Эти алгоритмы вычисляют сумму, объединение, пересечение и т.д.

Слияние двух упорядоченных множеств

```
OutputIterator
merge (InputIterator source1Beg, InputIterator source1End,
       InputIterator source2Beg, InputIterator source2End,
       OutputIterator destBeg)

OutputIterator
merge (InputIterator source1Beg, InputIterator source1End,
       InputIterator source2Beg, InputIterator source2End,
       OutputIterator destBeg, BinaryPredicate op)
```

- Обе формы выполняют слияние элементов упорядоченных диапазонов-источников [*source1Beg*,*source1End*) и [*source2Beg*,*source2End*), так что диапазон-получатель, начинающийся с позиции `destBeg`, содержит все элементы, содержащиеся в первом диапазоне-источнике, и все элементы, содержащиеся во втором диапазоне-источнике. Например, вызов алгоритма `merge()` для последовательностей

  ```
  1 2 2 4 6 7 7 9
  ```

  и

  ```
  2 2 2 3 6 668 9
  ```

  вычисляет следующий результат:

  ```
  1 2 2 2 2 2 3 4 6 6 6 7 7 8 9 9
  ```

- Все элементы в диапазоне-получателе упорядочены.

- Обе формы возвращают позицию, расположенную за последним скопированным элементом в диапазоне-получателе (позиция первого элемента, который не был заменен).
- Аргумент *op* — это необязательный бинарный предикат, используемый в качестве критерия сортировки:

  op(*elem1*, *elem2*)
- Диапазоны-источники остаются неизменными.
- В соответствии со стандартом вызывающая сторона должна гарантировать, что оба диапазона-источника упорядочены изначально. Однако в большинстве реализаций этот алгоритм также объединяет два неупорядоченных диапазона-источника в неупорядоченный диапазон-получатель. Тем не менее, чтобы обеспечить переносимость программ, для неупорядоченных диапазонов следует дважды вызывать алгоритм `copy()`, а не алгоритм `merge()`.
- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Диапазон-получатель не должен перекрываться с диапазонами-источниками.
- Списки и последовательные списки имеют специальную функцию-член `merge()`, предназначенную для слияния двух списков (см. раздел 8.8.1).
- Для того чтобы элементы, общие для обоих диапазонов-источников, включались в диапазон-получатель только один раз, следует использовать алгоритм `set_union()`.
- Для обработки только тех элементов, которые принадлежат обоим диапазонам-источникам, следует использовать алгоритм `set_intersection()`.
- Сложность: линейная (не более *numElems1*+*numElems2*-1 сравнений).

Следующий пример демонстрирует использование алгоритма `merge()`:

```
// algo/merge1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   list<int> coll1;
   set<int> coll2;

   // заполняем коллекции упорядоченными элементами
   INSERT_ELEMENTS(coll1,1,6);
   INSERT_ELEMENTS(coll2,3,8);
   PRINT_ELEMENTS(coll1,"coll1: ");
   PRINT_ELEMENTS(coll2,"coll2: ");

   // выводим объединенную последовательность
   cout << "merged: ";
   merge (coll1.cbegin(), coll1.cend(),
          coll2.cbegin(), coll2.cend(),
```

```
        ostream_iterator<int>(cout," "));
    cout << endl;
}
```

Программа выводит следующий результат:

```
coll1: 1 2 3 4 5 6
coll2: 3 4 5 6 7 8
merged: 1 2 3 3 4 4 5 5 6 6 7 8
```

В разделе "Примеры всех алгоритмов слияния" будет приведен другой пример, демонстрирующий отличие разных алгоритмов, предназначенных для объединения упорядоченных последовательностей.

Объединение двух упорядоченных последовательностей

```
OutputIterator
set_union (InputIterator source1Beg, InputIterator source1End,
           InputIterator source2Beg, InputIterator source2End,
           OutputIterator destBeg)

OutputIterator
set_union (InputIterator source1Beg, InputIterator source1End,
           InputIterator source2Beg, InputIterator source2End,
           OutputIterator destBeg, BinaryPredicate op)
```

- Обе формы выполняют слияние элементов упорядоченных диапазонов-источников [*source1Beg*,*source1End*) и [*source2Beg*,*source2End*), так что диапазон-получатель, начинающийся с позиции `destBeg`, содержит все элементы, содержащиеся в первом диапазоне-источнике, все элементы, содержащиеся во втором диапазоне-источнике, или все элементы, содержащиеся в обоих диапазонах. Например, вызов алгоритма `set_union()` для последовательностей

  ```
  1 2 2 4 6 7 7 9
  ```

  и

  ```
  2 2 2 3 6 6 8 9
  ```

  приводит к результату

  ```
  1 2 2 2 3 4 6 6 7 7 8 9
  ```

- Все элементы в диапазоне-получателе упорядочены.
- Элементы, принадлежащие обоим диапазонам, включаются в объединение только один раз. Однако дубликаты возможны, если элементы входят несколько раз в один из диапазонов-источников. Количество одинаковых элементов в диапазоне-получателе равно их максимальному количеству в диапазонах-источниках.
- Обе формы возвращают позицию, расположенную за последним скопированным элементом в диапазоне-получателе (позиция первого элемента, который не был заменен).
- Аргумент *op* — это необязательный бинарный предикат, используемый в качестве критерия сортировки:

  op(*elem1*, *elem2*)

- Диапазоны-источники остаются неизменными.
- Вызывающая сторона должна гарантировать, что диапазоны изначально упорядочены в соответствии с критерием сортировки.
- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Диапазон-получатель не должен перекрываться с диапазонами-источниками.
- Для того чтобы получить все элементы, общие для обоих диапазонов-источников, не исключая общие элементы, следует использовать алгоритм `merge()`.
- Сложность: линейная (не более `2*` (*numElems1*+*numElems2*) `-1` сравнений).

Пример использования алгоритма `set_union()` приведен в разделе "Примеры всех алгоритмов слияния". Он демонстрирует, чем отличаются разные алгоритмы, предназначенные для объединения упорядоченных последовательностей.

Пересечение двух упорядоченных множеств

```
OutputIterator
set_intersection (InputIterator source1Beg, InputIterator source1End,
                  InputIterator source2Beg, InputIterator source2End,
                  OutputIterator destBeg)

OutputIterator
set_intersection (InputIterator source1Beg, InputIterator source1End,
                  InputIterator source2Beg, InputIterator source2End,
                  OutputIterator destBeg, BinaryPredicate op)
```

- Обе формы выполняют слияние элементов упорядоченных диапазонов-источников [*source1Beg*,*source1End*) и [*source2Beg*,*source2End*), так что диапазон-получатель, начинающийся с позиции `destBeg`, содержит все элементы, содержащиеся в обоих диапазонах-источниках. Например, вызов алгоритма `set_intersection()` для последовательностей

  ```
  1 2 2 4 6 7 7 9
  ```

  и

  ```
  2 2 2 3 6 6 8 9
  ```

  приводит к результату

  ```
  2 2 6 9
  ```

- Все элементы в диапазоне-получателе упорядочены.
- Дубликаты возможны, если элементы входят в оба диапазона-источника несколько раз. Количество одинаковых элементов в диапазоне-источнике равно их минимальному количеству в обоих диапазонах-источниках.
- Обе формы возвращают позицию, расположенную за последним элементом в диапазоне-получателе, полученном после слияния.
- Аргумент *op* — это необязательный бинарный предикат, используемый в качестве критерия сортировки:

  op (*elem1*, *elem2*)

- Диапазоны-источники остаются неизменными.
- Вызывающая сторона должна гарантировать, что диапазоны изначально упорядочены в соответствии с критерием сортировки.
- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Диапазон-получатель не должен перекрываться с диапазонами-источниками.
- Сложность: линейная (не более `2*` (*numElems1*+*numElems2*) `-1` сравнений).

Пример использования алгоритма `set_intersection()` приведен в разделе "Примеры всех алгоритмов слияния". Он демонстрирует, чем отличаются разные алгоритмы, предназначенные для объединения упорядоченных последовательностей.

Разность между двумя упорядоченными множествами

```
OutputIterator
set_difference (InputIterator source1Beg, InputIterator source1End,
                InputIterator source2Beg, InputIterator source2End,
                OutputIterator destBeg)

OutputIterator
set_difference (InputIterator source1Beg, InputIterator source1End,
                InputIterator source2Beg, InputIterator source2End,
                OutputIterator destBeg, BinaryPredicate op)
```

- Обе формы выполняют слияние элементов упорядоченных диапазонов-источников [*source1Beg*,*source1End*) и [*source2Beg*,*source2End*), так что диапазон-получатель, начинающийся с позиции `destBeg`, содержит все элементы, содержащиеся в первом, но не во втором диапазоне-источнике. Например, вызов алгоритма `set_difference()` для последовательностей

  ```
  1 2 2 4 6 7 7 9
  ```

  и

  ```
  2 2 2 3 6 6 8 9
  ```

  приводит к результату

  ```
  1 4 7 7
  ```

- Все элементы в диапазоне-получателе упорядочены.
- Дубликаты возможны, если элементы входят в первый диапазон-источник несколько раз. Количество одинаковых элементов в диапазоне-источнике равно разности между их количеством в первом и втором диапазонах-источниках. Если количество одинаковых элементов во втором диапазоне-источнике больше, чем в первом, то их количество в диапазоне-получателе устанавливается равным нулю.
- Обе формы возвращают позицию, расположенную за последним элементом в диапазоне-получателе, полученном после слияния.
- Аргумент *op* — это необязательный бинарный предикат, используемый в качестве критерия сортировки:

  op (*elem1*, *elem2*)

- Диапазоны-источники остаются неизменными.
- Вызывающая сторона должна гарантировать, что диапазоны изначально упорядочены в соответствии с критерием сортировки.
- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Диапазон-получатель не должен перекрываться с диапазонами-источниками.
- Сложность: линейная (не более `2*` (*numElems1*+*numElems2*) `-1` сравнений).

Пример использования алгоритма `set_difference()` приведен в разделе "Примеры всех алгоритмов слияния". Он демонстрирует, чем отличаются разные алгоритмы, предназначенные для объединения упорядоченных последовательностей.

```
OutputIterator
set_symmetric_difference(InputIterator source1Beg, InputIterator source1End,
                         InputIterator source2Beg, InputIterator source2End,
                         OutputIterator destBeg)
OutputIterator
set_symmetric_difference(InputIterator source1Beg, InputIterator source1End,
                         InputIterator source2Beg, InputIterator source2End,
                         OutputIterator destBeg, BinaryPredicate op)
```

- Обе формы выполняют слияние элементов упорядоченных диапазонов-источников [*source1Beg*,*source1End*) и [*source2Beg*,*source2End*), так что диапазон-получатель, начинающийся с позиции `destBeg`, содержит все элементы, содержащиеся либо в первом, либо во втором диапазоне-источнике, но не в обоих одновременно. Например, вызов алгоритма `set_symmetric_difference()` для последовательностей

  ```
  1 2 2 4 6 7 7 9
  ```

  и

  ```
  2 2 2 3 6 6 8 9
  ```

  приводит к результату

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

- Все элементы в диапазоне-получателе упорядочены.
- Дубликаты возможны, если элементы несколько раз входят в один из диапазонов-источников. Количество одинаковых элементов в диапазоне-источнике равно разности между их количеством в первом и во втором диапазонах-источниках.
- Обе формы возвращают позицию, расположенную за последним элементом в диапазоне-получателе, полученном после слияния.
- Аргумент *op* — это необязательный бинарный предикат, используемый в качестве критерия сортировки:

  op (*elem1*, *elem2*)

- Диапазоны-источники остаются неизменными.
- Вызывающая сторона должна гарантировать, что диапазоны изначально упорядочены в соответствии с критерием сортировки.

- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Диапазон-получатель не должен перекрываться с диапазонами-источниками.
- Сложность: линейная (не более 2* (*numElems1+numElems2*) -1 сравнений).

Пример использования алгоритма `set_symmetric_difference()` приведен в разделе "Примеры всех алгоритмов слияния". Он демонстрирует, чем отличаются разные алгоритмы, предназначенные для объединения упорядоченных последовательностей.

Пример всех алгоритмов слияния

Следующий пример сравнивает разные алгоритмы, выполняющие слияние элементов двух упорядоченных диапазонов-источников, демонстрируя их работу и различия:

```
// algo/sorted1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> c1 = { 1, 2, 2, 4, 6, 7, 7, 9 };
   deque<int> c2 = { 2, 2, 2, 3, 6, 6, 8, 9 };

   // выводим диапазоны-источники
   cout << "c1:                          ";
   copy (c1.cbegin(), c1.cend(),
         ostream_iterator<int>(cout," "));
   cout << endl;
   cout << "c2:                          ";
   copy (c2.cbegin(), c2.cend(),
   ostream_iterator<int>(cout," "));
   cout << '\n' << endl;

   // выполняем слияние диапазонов с помощью алгоритма merge()
   cout << "merge():                     ";
   merge (c1.cbegin(), c1.cend(),
          c2.cbegin(), c2.cend(),
          ostream_iterator<int>(cout," "));
   cout << endl;

   // объединяем диапазоны с помощью алгоритма set_union()
   cout << "set_union():                 ";
   set_union (c1.cbegin(), c1.cend(),
              c2.cbegin(), c2.cend(),
              ostream_iterator<int>(cout," "));
   cout << endl;

   // вычисляем пересечение диапазонов с помощью алгоритма set_intersection()
   cout << "set_intersection():          ";
   set_intersection (c1.cbegin(), c1.cend(),
                     c2.cbegin(), c2.cend(),
```

```
                    ostream_iterator<int>(cout," "));
    cout << endl;

    // определяем элементы первого диапазона, не входящие во второй диапазон,
    // с помощью алгоритма set_difference()
    cout << "set_difference():           ";
    set_difference (c1.cbegin(), c1.cend(),
                    c2.cbegin(), c2.cend(),
                    ostream_iterator<int>(cout," "));
    cout << endl;

    // вычисляем разность диапазонов с помощью алгоритма
    // set_symmetric_difference()
    cout << "set_symmetric_difference(): ";
    set_symmetric_difference (c1.cbegin(), c1.cend(),
                              c2.cbegin(), c2.cend(),
                              ostream_iterator<int>(cout," "));
    cout << endl;
}
```

Программа выдает следующий результат:

```
c1:                          1 2 2 4 6 7 7 9
c2:                          2 2 2 3 6 6 8 9

merge():                     1 2 2 2 2 2 3 4 6 6 6 7 7 8 9 9
set_union():                 1 2 2 2 3 4 6 6 7 7 8 9
set_intersection():          2 2 6 9
set_difference():            1 4 7 7
set_symmetric_difference(): 1 2 3 4 6 7 7 8
```

Слияние смежных упорядоченных диапазонов

```
void
inplace_merge (BidirectionalIterator beg1,BidirectionalIterator end1beg2,
               BidirectionalIterator end2)

void
inplace_merge (BidirectionalIterator beg1,BidirectionalIterator end1beg2,
               BidirectionalIterator end2,BinaryPredicate op)
```

- Обе формы выполняют слияние смежных упорядоченных диапазонов-источников [*beg1*,*end1beg2*) и [*end1beg2*,*end2*), так что диапазон [*beg1*,*end2*) содержит элементы обоих диапазонов.
- Сложность: линейная (*numElems*-1 сравнений), если есть достаточное количество памяти, или *n*log(n)* в противном случае (*numElems**log(*numElems*) сравнений).

Следующая программа демонстрирует использование алгоритма inplace_merge():

```
// algo/inplacemerge1.cpp

#include "algostuff.hpp"
using namespace std;
```

```
int main()
{
   list<int> coll;

   // вставляем две упорядоченные последовательности
   INSERT_ELEMENTS(coll,1,7);
   INSERT_ELEMENTS(coll,1,8);
   PRINT_ELEMENTS(coll);

   // находим начало второй части (элемент после 7)
   list<int>::iterator pos;
   pos = find (coll.begin(), coll.end(),    // диапазон
               7);                          // значение
   ++pos;

   // выполняем слияние в один упорядоченный диапазон
   inplace_merge (coll.begin(), pos, coll.end());
   PRINT_ELEMENTS(coll);
}
```

Программа выдает следующий результат:

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

11.11. Численные алгоритмы

В разделе описываются алгоритмы библиотеки STL, предназначенные для работы с числами. Однако с их помощью можно обрабатывать не только числа. Например, алгоритм `accumulate()` позволяет вычислить сумму нескольких строк. Для использования численных алгоритмов в программу необходимо включить заголовочный файл `<numeric>`.

```
#include <numeric>
```

11.11.1. Вычисления

Вычисление результата по одной последовательности

```
T
accumulate (InputIterator beg, InputIterator end,
            T initValue)

T
accumulate (InputIterator beg, InputIterator end,
            T initValue, BinaryFunc op)
```

- Первая форма вычисляет и возвращает сумму значения *initValue* и всех элементов диапазона [*beg*,*end*). В частности, она применяет к каждому элементу следующую операцию:

```
initValue = initValue + elem
```

- Вторая форма вычисляет и возвращает результат применения операции *op* к значению *initValue* и всем элементам диапазона [*beg*,*end*). В частности, она применяет к каждому элементу следующую операцию:

  ```
  initValue = op(initValue,elem)
  ```

- Таким образом, для значений

  ```
  a1 a2 a3 a4 ...
  ```

  алгоритм вычисляет и возвращает либо значение

  ```
  initValue + a1 + a2 + a3 + ...
  ```

  либо

  ```
  initValue op a1 op a2 op a3 op ...
  ```

  соответственно.

- Если диапазон пустой (*beg*==*end*), то обе формы возвращают значение *initValue*.
- Операция *op* не должна модифицировать передаваемые аргументы.
- Сложность: линейная (*numElems* вызовов оператора + или *op*() соответственно).

Следующая программа демонстрирует использование алгоритма `accumulate()` для вычисления суммы и произведения всех элементов диапазона:

```
// algo/accumulate1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll;

   INSERT_ELEMENTS(coll,1,9);
   PRINT_ELEMENTS(coll);

   // вычисляем сумму элементов
   cout << "sum: "
        << accumulate (coll.cbegin(), coll.cend(), // диапазон
                       0)                          // начальное значение
        << endl;

   // вычисляем сумму элементов, которые меньше 100
   cout << "sum: "
        << accumulate (coll.cbegin(), coll.cend(), // диапазон
                       -100)                       // начальное значение
        << endl;

   // вычисляем произведение элементов
   cout << "product: "
        << accumulate (coll.cbegin(), coll.cend(), // диапазон
                       1,                          // начальное значение
                       multiplies<int>())          // операция
        << endl;
```

```
    // вычисляем произведение элементов (используем 0 как начальное значение)
    cout << "product: "
         << accumulate (coll.cbegin(), coll.cend(), // диапазон
                        0,                          // начальное значение
                        multiplies<int>())          // операция
         << endl;
}
```

Программа выдает следующий результат:

```
1 2 3 4 5 6 7 8 9
sum: 45
sum: -55
product: 362880
product: 0
```

Последнее вычисленное значение равно 0, потому что произведение любого значения и нуля равно нулю.

Вычисление скалярного произведения двух последовательностей

```
T
inner_product (InputIterator1 beg1, InputIterator1 end1,
               InputIterator2 beg2, T initValue)

T
inner_product (InputIterator1 beg1, InputIterator1 end1,
               InputIterator2 beg2, T initValue,
               BinaryFunc op1, BinaryFunc op2)
```

- Первая форма вычисляет и возвращает сумму значения *initValue* и скалярного произведения элементов диапазона [*beg*,*end*) и диапазона, начинающегося с позиции *beg2*. В частности, она применяет к каждому элементу следующую операцию:

  ```
  initValue = initValue + elem1 * elem2
  ```

- Вторая форма вычисляет и возвращает результат применения операции *op* к значению *initValue* и всем элементам диапазона [*beg*,*end*) и диапазона, начинающегося с позиции *beg2*. В частности, она применяет ко всем парам соответствующих элементов операции *op1* и *op2*:

  ```
  6initValue = op1(initValue,op2(elem1,elem2))
  ```

- Таким образом, для значений

  ```
  a1 a2 a3 ...
  b1 b2 b3 ...
  ```

  алгоритм вычисляет и возвращает либо значение

  ```
  initValue + (a1 * b1) + (a2 * b2) + (a3 * b3) + ...
  ```

  либо

  ```
  initValue op1 (a1 op2 b1) op1 (a2 op2 b2) op1 (a3 op2 b3) op1 ...
  ```

  соответственно.

- Если диапазон пустой (*beg==end*), то обе формы возвращают значение *initValue*.
- Вызывающая сторона должна гарантировать, что диапазон, начинающийся с позиции *beg2*, содержит достаточное количество элементов
- Операции *op1* и *op2* не должны модифицировать свои аргументы.
- Сложность: линейная (*numElems* вызовов операторов + и * или *numElems* вызовов операций *op1*() и *op2*() соответственно).

Следующая программа демонстрирует использование алгоритма `inner_product()`. Она вычисляет сумму произведений и произведение сумм двух последовательностей:

```
// algo/innerproduct1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   list<int> coll;

   INSERT_ELEMENTS(coll,1,6);
   PRINT_ELEMENTS(coll);

   // вычисляем сумму всех произведений
   // (0 + 1*1 + 2*2 + 3*3 + 4*4 + 5*5 + 6*6)
   cout << "inner product: "
        << inner_product (coll.cbegin(), coll.cend(),  // первый диапазон
                          coll.cbegin(),               // второй диапазон
                          0)                           // начальное значение
        << endl;

   // вычисляем сумму 1*6 ... 6*1
   // (0 + 1*6 + 2*5 + 3*4 + 4*3 + 5*2 + 6*1)
   cout << "inner reverse product: "
        << inner_product (coll.cbegin(), coll.cend(),  // первый диапазон
                          coll.crbegin(),              // второй диапазон
                          0)                           // начальное значение
        << endl;

   // вычисляем произведение всех сумм
   // (1 * 1+1 * 2+2 * 3+3 * 4+4 * 5+5 * 6+6)
   cout << "product of sums: "
        << inner_product (coll.cbegin(), coll.cend(),  // первый диапазон
                          coll.cbegin(),               // второй диапазон
                          1,                           // начальное значение
                          multiplies<int>(),           // внешняя операция
                          plus<int>())                 // внутренняя операция
        << endl;
}
```

Программа выдает следующий результат:

```
1 2 3 4 5 6
inner product: 91
inner reverse product: 56
product of sums: 46080
```

11.11.2. Преобразования относительных и абсолютных значений

Следующие два алгоритма позволяют преобразовывать последовательность относительных значений в последовательность абсолютных значений, и наоборот.

Преобразование относительных значений в абсолютные

```
OutputIterator
partial_sum (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg)

OutputIterator
partial_sum (InputIterator sourceBeg, InputIterator sourceEnd,
             OutputIterator destBeg, BinaryFunc op)
```

- Первая форма вычисляет частичную сумму для каждого элемента диапазона-источника [*sourceBeg*, *sourceEnd*) и записывает каждый результат в диапазон-получатель, начиная с позиции *destBeg*.
- Вторая форма применяет операцию *op* к каждому элементу диапазона-источника [*sourceBeg*,*sourceEnd*), объединяет результат со всеми предыдущими значениями и записывает в диапазон-получатель, начинающийся с позиции *destBeg*.
- Таким образом, для значений

  ```
  a1 a2 a3 ...
  ```

  алгоритм вычисляет либо

  ```
  a1, a1 + a2, a1 + a2 + a3, ...
  ```

  либо

  ```
  a1, a1 op a2, a1 op a2 op a3, ...
  ```

  соответственно.
- Оба алгоритма возвращают позицию, находящуюся за последним скопированным элементом в диапазоне-получателе (последний элемент, который не был заменен).
- Первая форма эквивалентна преобразованию последовательности относительных значений в последовательность абсолютных значений. В этом смысле алгоритм `partial_sum()` является дополнением алгоритма `adjacent_difference()`.
- Источник и получатель могут быть идентичными.
- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Операция *op* не должна модифицировать передаваемые аргументы.
- Сложность: линейная (*numElems* вызовов оператора + или *op*() соответственно).

Следующая программа демонстрирует несколько примеров использования алгоритмов `partial_sum()`:

```
// algo/partialsum1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll;

   INSERT_ELEMENTS(coll,1,6);
   PRINT_ELEMENTS(coll);

   // выводим все частичные суммы
   partial_sum (coll.cbegin(), coll.cend(),          // диапазон-источник
                ostream_iterator<int>(cout," "));    // диапазон-получатель
                cout << endl;

   // выводим все частичные произведения
   partial_sum (coll.cbegin(), coll.cend(),          // диапазон-источник
                ostream_iterator<int>(cout," "),     // диапазон-получатель
                multiplies<int>());                  // операция
   cout << endl;
}
```

Программа выводит следующий результат:

```
1 2 3 4 5 6
1 3 6 10 15 21
1 2 6 24 120 720
```

Пример преобразования относительных значений в абсолютные приведен ниже.

Преобразование абсолютных значений в относительные

```
OutputIterator
adjacent_difference (InputIterator sourceBeg, InputIterator sourceEnd,
                     OutputIterator destBeg)

OutputIterator
adjacent_difference (InputIterator sourceBeg, InputIterator sourceEnd,
                     OutputIterator destBeg, BinaryFunc op)
```

- Первая форма вычисляет разность между каждым элементом диапазона [*sourceBeg*,*sourceEnd*) и его предшественником и записывает результат в диапазон-получатель, начинающийся с позиции *destBeg*.
- Вторая форма применяет операцию *op* к каждому элементу в диапазоне [*sourceBeg*,*sourceEnd*) и его предшественнику и записывает результат в диапазон-получатель, начинающийся с позиции *destBeg*.
- Первый элемент только копируется.

- Таким образом, для значений

  `a1 a2 a3 a4 ...`

  алгоритм вычисляет и записывает либо

  `a1, a2 - a1, a3 - a2, a4 - a3, ...`

  либо

  `a1, a2` *op* `a1, a3` *op* `a2, a4` *op* `a3, ...`

  соответственно.
- Обе формы возвращают позицию, находящуюся за последним записанным элементом в диапазоне-получателе (позиция первого элемента, который не был переписан).
- Первая форма эквивалентна преобразованию последовательности абсолютных значений в относительные. В этом смысле алгоритм `adjacent_difference()` является дополнением алгоритма `partial_sum()`.
- Источник и получатель могут совпадать.
- Вызывающая сторона должна гарантировать, что диапазон-получатель достаточно большой или что используются итераторы вставки.
- Операция *op* не должна модифицировать передаваемые аргументы.
- Сложность: линейная (*numElems*-1 вызовов оператора - или *op*() соответственно).

Следующая программа демонстрирует несколько примеров использования алгоритма `adjacent_difference()`:

```
// algo/adjacentdiff1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   deque<int> coll;

   INSERT_ELEMENTS(coll,1,6);
   PRINT_ELEMENTS(coll);

   // выводим все разности между элементами
   adjacent_difference (coll.cbegin(), coll.cend(),        // источник
                        ostream_iterator<int>(cout," "));  // получатель
   cout << endl;

   // выводим все суммы с предшественниками
   adjacent_difference (coll.cbegin(), coll.cend(),        // источник
                        ostream_iterator<int>(cout," "),   // получатель
                        plus<int>());                      // операция
   cout << endl;

   // выводим все произведения элементов
   adjacent_difference (coll.cbegin(), coll.cend(),        // источник
```

```
                        ostream_iterator<int>(cout," "),    // получатель
                        multiplies<int>());                 // операция
   cout << endl;
}
```

Программа выводит следующий результат:

```
1 2 3 4 5 6
1 1 1 1 1 1
1 3 5 7 9 11
1 2 6 12 20 30
```

Пример преобразования относительных величин в абсолютные, и наоборот, приводится в следующем подразделе.

Пример преобразования относительных величин в абсолютные

Следующий пример демонстрирует использование алгоритмов `partial_sum()` и `adjacent_difference()` для преобразования последовательности относительных величин в абсолютные, и наоборот.

```
// algo/relabs1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
   vector<int> coll = { 17, -3, 22, 13, 13, -9 };
   PRINT_ELEMENTS(coll,"coll:     ");

   // преобразовываем в относительные величины
   adjacent_difference (coll.cbegin(), coll.cend(),    // источник
                        coll.begin());                 // получатель
   PRINT_ELEMENTS(coll,"relative: ");

   // преобразовываем в абсолютные величины
   partial_sum (coll.cbegin(), coll.cend(),            // источник
                coll.begin());                         // получатель
   PRINT_ELEMENTS(coll,"absolute: ");
}
```

Программа выводит следующий результат:

```
coll:     17 -3 22 13 13 -9
relative: 17 -20 25 -9 0 -22
absolute: 17 -3 22 13 13 -9
```

Глава 12

Специальные контейнеры

Стандартная библиотека C++ содержит не только контейнеры STL, но и контейнеры, предназначенные для особых целей и предоставляющие простые и понятные интерфейсы. Эти контейнеры можно разделить на так называемые *контейнерные адаптеры*, которые приспосабливают стандартные контейнеры STL для особых целей, и битовые множества, представляющие собой контейнеры, содержащие биты или булевы величины.

Существуют три стандартных контейнерных адаптера: стеки, очереди и очереди с приоритетами. В очередях с приоритетами элементы сортируются автоматически в соответствии с критерием сортировки. Таким образом, следующий элемент очереди с приоритетами — это элемент, имеющий наивысшее значение.

Битовое множество — это битовое поле с произвольным, но фиксированным количеством битов. Отметим, что в стандартной библиотеке C++ есть специальный контейнер с переменным размером для хранения булевых значений: `vector<bool>` (см. раздел 7.3.6).

Недавние изменения, связанные со стандартом C++11

Практически все функциональные возможности контейнерных адаптеров были описаны еще в стандарте C++98. Перечислим наиболее важные функциональные возможности, добавленные стандартом C++11.

- Контейнерные адаптеры содержат определеия типов `reference` и `const_reference` (см. раздел 12.4.1).
- Контейнерные адаптеры поддерживают семантику перемещения и rvalue-ссылки:
  - функция `push()` обеспечивает семантику перемещения (см. разделы 12.1.2 и 12.4.4);
  - исходный контейнер может быть перемещен (см. раздел 12.4.2).
- Контейнерные адаптеры имеют функцию-член `emplace()`, автоматически создающую новый элемент, инициализированный передаваемыми аргументами (см. разделы 12.1.2 и 12.4.4).
- Контейнерные адаптеры имеют функцию-член `swap()` (см. раздел 12.4.4).
- Контейнерные адаптеры позволяют передавать своим конструкторам специальный механизм распределения памяти (см. раздел 12.4.2).

12.1. Стеки

Класс `stack<>` реализует стек (известный как LIFO). С помощью функции-члена `push()` в стек можно вставлять любое количество элементов (рис. 12.1). С помощью

функции-члена pop() вставленные ранее элементы можно удалять из стека в обратном порядке ("последним вошел, первым вышел").

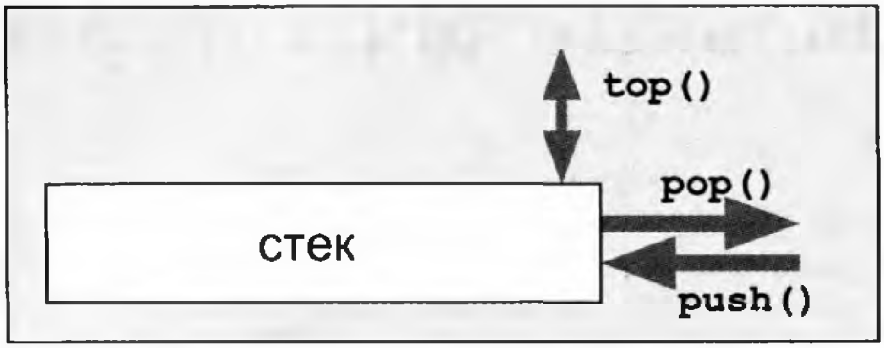


Рис. 12.1. Интерфейс стека

Для использования стека в программу следует включить заголовочный файл <stack>.

```
#include <stack>
```

В заголовочном файле <stack> класс stack определен следующим образом:

```
namespace std {
   template <typename T,
             typename Container = deque<T>>
             class stack;
}
```

Первый шаблонный параметр — это тип элементов. Необязательный второй параметр определяет контейнер, который стек будет использовать для хранения своих элементов. По умолчанию в качестве такого контейнера используется дек. Этот выбор объясняется тем, что в отличие от векторов деки освобождают память при удалении элементов и не копируют все элементы при повторном выделении памяти (см. раздел 7.12, в котором рассматривается вопрос о том, когда и какой контейнер целесообразно использовать).

Например, следующее объявление определяет стек целых чисел:

```
std::stack<int> st; // стек целых чисел
```

Реализация стека просто отображает его операции в соответствующие вызовы контейнера, который внутренне используется стеком (рис. 12.2). Можно использовать любой последовательный контейнерный класс, имеющий функции-члены back(), push_back() и pop_back(). Например, в качестве контейнера можно использовать вектор или список.

```
std::stack<int,std::vector<int>> st; // стек целых чисел, использующий вектор
```

12.1.1. Основной интерфейс

Основной интерфейс стека состоит из функций-членов push(), top() и pop().

- Функция **push()** вставляет элемент в стек.
- Функция **top()** возвращает элемент на вершине стека.
- Функция **pop()** удаляет элемент из стека.

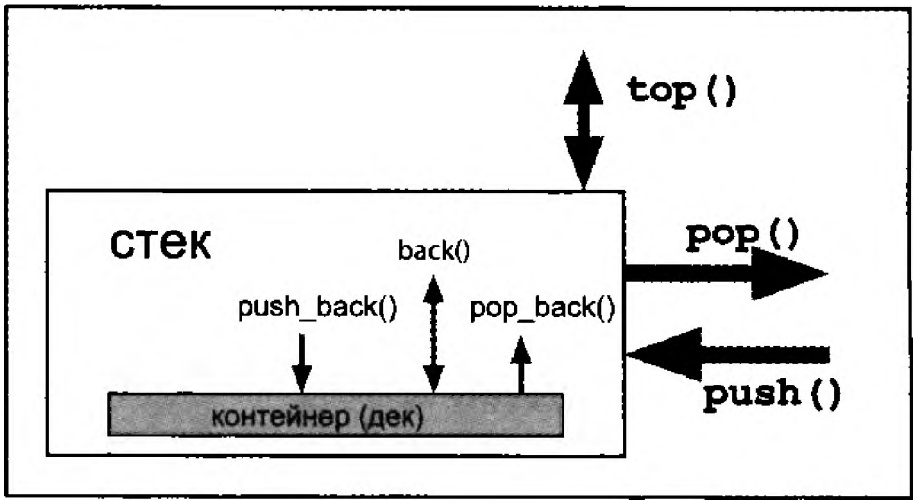


Рис. 12.2. Внутренний интерфейс стека

Отметим, что функция-член `pop()` удаляет следующий элемент, но не возвращает его, а функция-член `top()` возвращает следующий доступный элемент стека, не удаляя его. Таким образом, для обработки и удаления следующего элемента стека следует вызывать обе эти функции. Этот интерфейс несколько неудобен, но он работает лучше, если нужно только удалить следующий элемент, не обрабатывая его[1]. Отметим, что поведение функций-членов `top()` и `pop()` для пустого стека не определено. Функции-члены `size()` и `empty()` проверяют, содержит ли стек элементы.

Если вам не нравится стандартный интерфейс класса `stack<>`, можно легко написать более удобный интерфейс. Пример такого интерфейса приведен в разделе 12.1.3.

12.1.2. Пример использования стеков

Следующая программа демонстрирует класс `stack<>`:

```
// contadapt/stack1.cpp

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

int main()
{
   stack<int> st;

   // вносим в стек три элемента
   st.push(1);
   st.push(2);
   st.push(3);

   // выводим на печать и удаляем два элемента из стека
   cout << st.top() << ' ';
```

[1] Как указано в книге Г. Саттер *Решение сложных задач на C++* (М.: И.Д. "Вильямс", 2002), функция, не только удаляющая элемент с вершины стека, но при этом и возвращающая его, оказывается небезопасной в смысле исключений. — *Примеч. консульт.*

```
    st.pop();
    cout << st.top() << ' ';
    st.pop();

    // модифицируем элемент на вершине
    st.top() = 77;

    // вносим два новых элемента
    st.push(4);
    st.push(5);

    // удаляем один элемент, не обрабатывая его
    st.pop();

    // выводим на печать и удаляем остальные элементы
    while (!st.empty()) {
        cout << st.top() << ' ';
        st.pop();
    }
    cout << endl;
}
```

Эта программа выводит следующий результат:

```
3 2 4 77
```

Отметим, что при работе с нетривиальными типами элементов целесообразно рассмотреть возможность использования функции-члена `std::move()` для вставки элементов, которые больше не нужны, и функции-члена `emplace()` для автоматического создания элемента в стеке (обе функции появились вместе со стандартом C++11).

```
stack<pair<string,string>> st;

auto p = make_pair("hello","world");
st.push(move(p)); // OK, если p больше не используется

st.emplace("nico","josuttis");
```

12.1.3. Пользовательский класс стека

Стандартный класс `stack<>` пожертвовал удобством и безопасностью в пользу быстродействия. Мне это не нравится, поэтому я написал свой собственный класс стека, имеющий два преимущества.

1. Функция-член `pop()` не только удаляет элемент с вершины стека, но и возвращает его.
2. Функции-члены `pop()` и `top()` генерируют исключение, если стек пуст.

Кроме того, я не стал включать в класс функции, которые не нужны рядовому пользователю, например операции сравнения. Мой класс стека определен следующим образом:

```
// contadapt/Stack.hpp
/* ************************************************************
* Stack.hpp
* - более безопасный и удобный класс стека
* ************************************************************/
#ifndef STACK_HPP
#define STACK_HPP

#include <deque>
#include <exception>
template <typename T>

class Stack {
protected:
   std::deque<T> c; // контейнер для элементов

   public:
   // класс исключения для функций-членов pop() и top() при пустом стеке
   class ReadEmptyStack : public std::exception {
      public:
         virtual const char* what() const throw() {
            return "read empty stack";
   }
};

   // количество элементов
   typename std::deque<T>::size_type size() const {
      return c.size();
   }

   // пустой ли стек?
   bool empty() const {
      return c.empty();
   }

   // вносим элемент в стек
   void push (const T& elem) {
      c.push_back(elem);
   }

   // удаляем элемент из стека и возвращаем его значение
   T pop () {
      if (c.empty()) {
          throw ReadEmptyStack();
      }
      T elem(c.back());
      c.pop_back();
      return elem;
   }

   // возвращаем значение элемента на вершине стека
   T& top () {
      if (c.empty()) {
```

```
            throw ReadEmptyStack();
        }
        return c.back();
    }
};

#endif /* STACK_HPP */
```

Для работы с этим классом предыдущий пример надо переписать следующим образом:

```
// contadapt/stack2.cpp

#include <iostream>
#include <exception>
#include "Stack.hpp"    // используем специальный стек класса class
using namespace std;

int main()
{
    try {
        Stack<int> st;

        // вносим три элемента в стек
        st.push(1);
        st.push(2);
        st.push(3);

        // удаляем и выводим на печать два элемента из стека
        cout << st.pop() << ' ';
        cout << st.pop() << ' ';

        // модифицируем элемент на вершине
        st.top() = 77;

        // добавляем новые элементы
        st.push(4);
        st.push(5);

        // убираем один элемент без обработки
        st.pop();

        // удаляем и выводим на печать три элемента
        // - ОШИБКА: одного элемента слишком много
        cout << st.pop() << ' ';
        cout << st.pop() << endl;
        cout << st.pop() << endl;
    }
    catch (const exception& e) {
        cerr << "EXCEPTION: " << e.what() << endl;
    }
}
```

Дополнительный заключительный вызов функции-члена pop() вызывает ошибку. В отличие от стандартного класса стека он генерирует исключение, а не приводит к неопределенному поведению. Программа выводит следующий результат:

```
3 2 4 77
EXCEPTION: read empty stack
```

12.1.4. Подробное описание класса stack<>

Интерфейс класса stack<> более или менее соответствует членам внутреннего контейнера. Рассмотрим пример:

```
namespace std {
   template <typename T, typename Container = deque<T>>
   class stack {
      public:
        typedef typename  Container::value_type      value_type;
        typedef typename  Container::reference       reference
        typedef typename  Container::const_reference const_reference;
        typedef typename  Container::size_type       size_type;
        typedef           Container                  container_type;
      protected:
        Container c; // контейнер
      public:
        bool        empty() const                  { return c.empty(); }
        size_type   size() const                   { return c.size(); }
        void        push(const value_type& x)      { c.push_back(x); }
        void        push(value_type&& x)           { c.push_back(move(x)); }
        void        pop()                          { c.pop_back(); }
        value_type& top()                          { return c.back(); }
        const value_type& top()                    const { return c.back(); }
        template <typename... Args>
        void emplace(Args&&... args) {
               c.emplace_back(std::forward<Args>(args)...); }
        void swap (stack& s) ... { swap(c,s.c); }
        ...
   };
}
```

Подробное описание функций-членов и операций приведено в разделе 12.4.

12.2. Очереди

Класс queue<> реализует очередь (также известную как FIFO). С помощью функции-члена push() в очередь можно вставить любое количество элементов (рис. 12.3). С помощью функции-члена pop() из очереди можно удалять элементы в том же порядке, в котором они были вставлены ("первым вошел, первым вышел"). Таким образом, очередь можно использовать в качестве классического буфера данных.

Для использования очереди в программу необходимо вставить заголовочный файл <queue>:

```
#include <queue>
```

Рис. 12.3. Интерфейс очереди

В заголовочном файле <queue> класс `queue` определен следующим образом:

```
namespace std {
   template <typename T,
             typename Container = deque<T>>
             class queue;
}
```

Первый шаблонный параметр — это тип элементов. Необязательный второй шаблонный параметр определяет контейнер, который очередь использует для хранения своих элементов. По умолчанию для этого используется дек.

Например, следующее объявление определяет очередь строк:

```
std::queue<std::string> buffer;     // очередь строк
```

Реализация очереди просто отображает операции в соответствующие вызовы контейнера, который используется в очереди (рис. 12.4). Можно использовать любой последовательный контейнерный класс, имеющий функции-члены `front()`, `back()`, `push_back()` и `pop_front()`. Например, в качестве контейнера можно использовать список:

```
std::queue<std::string,std::list<std::string>> buffer;
```

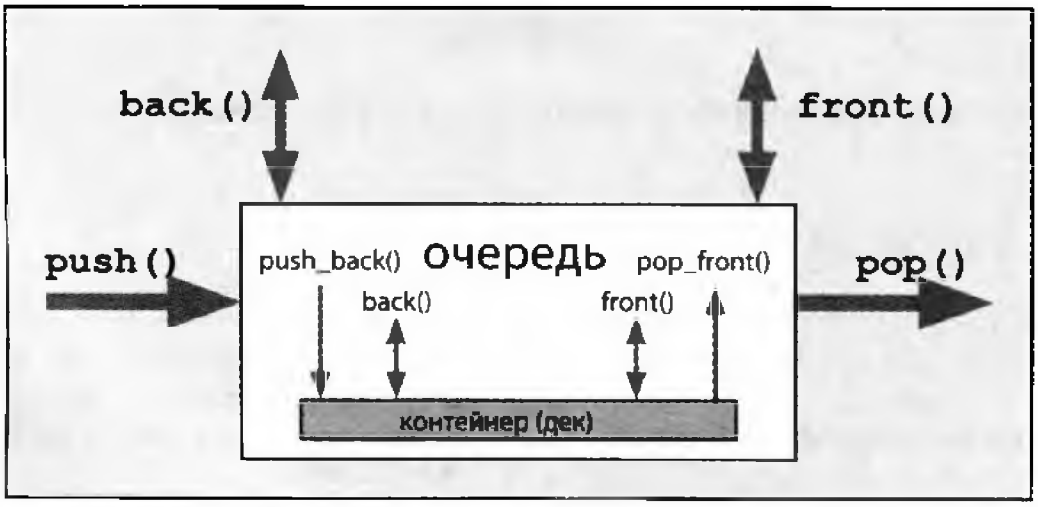


Рис. 12.4. Внутренний интерфейс очереди

12.2.1. Основной интерфейс

Основной интерфейс состоит из функций-членов `push()`, `front()`, `back()` и `pop()`.

- Функция-член **`push()`** вставляет элемент в очередь.
- Функция-член **`front()`** возвращает следующий доступный элемент в очереди (элемент, который был вставлен первым).
- Функция-член **`back()`** возвращает последний доступный элемент в очереди (элемент, который был вставлен последним).
- Функция-член **`pop()`** удаляет из очереди первый элемент.

Отметим, что функция-член `pop()` удаляет первый элемент, но не возвращает его, а функции-члены `front()` и `back()` возвращают элемент, не удаляя его. Таким образом, для обработки и удаления первого элемента очереди всегда следует вызывать функции-члены `front()` и `pop()`. Этот интерфейс несколько неудобный, но он работает лучше, если нужно только удалить следующий элемент, не обрабатывая его. Отметим, что поведение функций-членов `front()`, `back()` и `pop()` для пустой очереди не определено. Функции-члены `size()` и `empty()` проверяют, содержит ли очередь элементы.

Если вам не нравится стандартный интерфейс класса `queue<>`, можно легко написать более удобный интерфейс. Пример такого интерфейса приведен в разделе 12.2.3.

12.2.2. Пример использования очереди

Следующая программа демонстрирует использование класса `queue<>`:

```
// contadapt/queue1.cpp

#include <iostream>
#include <queue>
#include <string>
using namespace std;

int main()
{
   queue<string> q;

   // вставляем в очередь три элемента
   q.push("These ");
   q.push("are ");
   q.push("more than ");

   // читаем и выводим на печать два элемента из очереди
   cout << q.front();
   q.pop();
   cout << q.front();
   q.pop();

   // вставляем два новых элемента
   q.push("four ");
   q.push("words!");
```

```
    // пропускаем один элемент
    q.pop();

    // читаем, выводим на печать и удаляем из очереди два элемента
    cout << q.front();
    q.pop();
    cout << q.front() << endl;
    q.pop();

    // выводим на печать количество элементов в очереди
    cout << "number of elements in the queue: " << q.size()
         << endl;
}
```

Программа выводит следующий результат:

```
These are four words!
number of elements in the queue: 0
```

12.2.3. Пользовательский класс очереди

Стандартный класс `queue<>` пожертвовал удобством и безопасностью в пользу быстродействия. Это нравится не всем. Но можно легко написать свой собственный класс очереди, как показано выше при описании класса стека (раздел 12.1.3). Соответствующий пример представлен на веб-сайте, посвященном книге.

12.2.4. Подробное описание класса queue <>

Интерфейс класса `queue<>` более или менее соответствует членам внутреннего контейнера (см. раздел 12.1.4, в котором описаны соответствующие члены класса `stack<>`). Подробное описание членов и операций класса приведено в разделе 12.4.

12.3. Очереди с приоритетами

Класс `priority_queue<>` реализует очередь, из которой элементы считываются в соответствии с их приоритетом. Интерфейс такого класса похож на интерфейс очереди. Иначе говоря, функция-член `push()` вставляет элемент в очередь, а функции-члены `top()` и `pop()` обеспечивают доступ к следующему элементу и удаляют его (рис. 12.5). Однако следующий элемент — это не тот элемент, который был вставлен первым, а элемент, имеющий наивысший приоритет. Таким образом, элементы частично упорядочиваются по значениям. Как обычно, критерий сортировки можно задать в виде шаблонного параметра. По умолчанию элементы упорядочиваются с помощью оператора `<` в убывающем порядке.

Таким образом, следующий элемент — это всегда "наибольший" элемент. Если таких элементов несколько, то какой из них будет следующим, не определено.

Очереди с приоритетами определены в том же самом заголовочном файле, что и обычные очереди `<queue>`.

```
#include <queue>
```

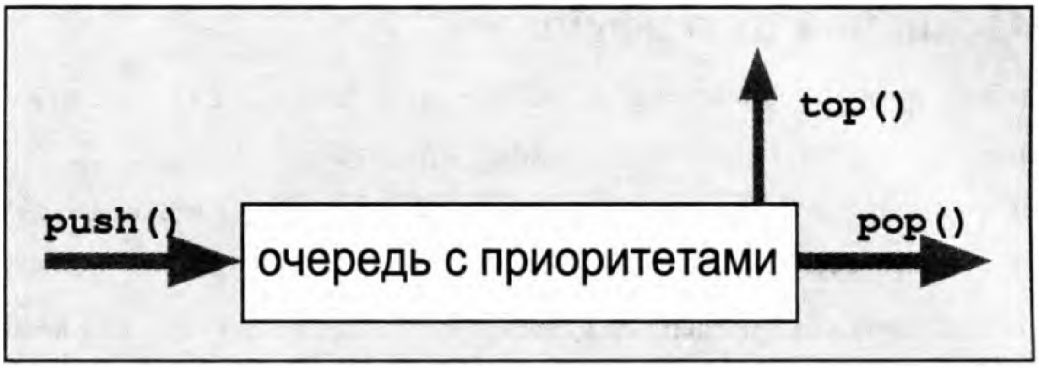


Рис. 12.5. Интерфейс очереди с приоритетами

В заголовочном файле <queue> класс `priority_queue` определен следующим образом:

```
namespace std {
   template <typename T,
             typename Container = vector<T>,
             typename Compare = less<typename Container::value_type>>
             class priority_queue;
}
```

Первый шаблонный параметр — это тип элементов. Необязательный второй шаблонный параметр определяет контейнер, который используется в очереди с приоритетами для хранения элементов. По умолчанию таким контейнером является вектор. Необязательный третий параметр определяет критерий сортировки, используемый для поиска следующего элемента с наивысшим приоритетом. По умолчанию этот параметр сравнивает элементы с помощью оператора <. Например, следующее определение объявляет очередь с приоритетами чисел типа `float`:

```
std::priority_queue<float> pbuffer; // очередь с приоритетами чисел типа float
```

Реализация очереди с приоритетами просто отображает операции в соответствующие вызовы контейнера, который используется в классе. Можно использовать любой последовательный контейнерный класс, имеющий итераторы произвольного доступа и функции-члены `front()`, `push_back()` и `pop_back()`. Произвольный доступ необходим для сортировки элементов, которая выполняется с помощью алгоритмов работы с пирамидами из библиотеки STL (см. раздел 11.9.4). Например, в качестве контейнера можно использовать дек:

```
std::priority_queue<float,std::deque<float>> pbuffer;
```

Для определения собственного критерия сортировки необходимо передать функцию, функциональный объект или лямбда-функцию в качестве бинарного предиката, используемого алгоритмами сортировки для сравнения двух элементов (более подробно критерии сортировки описаны в разделах 7.7.2 и 10.1.1). Например, следующее объявление определяет очередь с приоритетами с обратной сортировкой:

```
std::priority_queue<float,std::vector<float>,
                          std::greater<float>> pbuffer;
```

В этой очереди с приоритетами следующий элемент всегда является одним из наименьших.

12.3.1. Основной интерфейс

Основной интерфейс состоит из функций-членов `push()`, `top()` и `pop()`.

- Функция-член **`push()`** вставляет элемент в очередь.
- Функция-член **`top()`** возвращает следующий доступный элемент в очереди.
- Функция-член **`pop()`** удаляет элемент из очереди.

Как и во всех других контейнерных адаптерах, функция-член `pop()` удаляет следующий элемент, но не возвращает его, а функция-член `top()` возвращает элемент, не удаляя его. Таким образом, для обработки и удаления следующего элемента в очереди всегда следует вызывать обе эти функции-члены. Кроме того, как обычно, поведение функций-членов `top()` и `pop()` для пустой очереди не определено. Если есть сомнения, следует использовать функции-члены `size()` и `empty()`.

12.3.2. Пример использования очереди с приоритетами

Следующая программа демонстрирует использование класса `priority_queue<>`:

```
// contadapt/priorityqueue1.cpp

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

int main()
{
   priority_queue<float> q;

   // вставляем в очередь с приоритетами три элемента
   q.push(66.6);
   q.push(22.2);
   q.push(44.4);

   // читаем, выводим на печать и удаляем два элемента
   cout << q.top() << ' ';
   q.pop();
   cout << q.top() << endl;
   q.pop();

   // вставляем еще три элемента
   q.push(11.1);
   q.push(55.5);
   q.push(33.3);

   // пропускаем один элемент
   q.pop();

   // выводим на печать и удаляем остальные элементы
   while (!q.empty()) {
      cout << q.top() << ' ';
      q.pop();
```

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

Программа выводит следующий результат:

```
66.6 44.4
33.3 22.2 11.1
```

Как видим, после вставки чисел `66.6`, `22.2` и `44.4` программа выводит на экран числа `66.6` и `44.4` как старшие элементы. После вставки еще трех элементов очередь с приоритетами содержит числа `22.2`, `11.1`, `55.5` и `33.3` (в порядке вставки). Следующий элемент пропускается с помощью вызова функции-члена `pop()`, поэтому заключительный цикл выводит на экран элементы `33.3`, `22.2` и `11.1` в указанном порядке.

12.3.3. Подробное описание класса `priority_queue<>`

Очереди с приоритетами используют алгоритмы STL для работы с пирамидами.

```
namespace std {
   template <typename T, typename Container = vector<T>,
             typename Compare = less<typename Container::value_type>>
   class priority_queue {
      protected:
         Compare comp;    // критерий сортировки
         Container c;     // контейнер
   public:
      // конструкторы
      explicit priority_queue(const Compare& cmp = Compare(),
                              const Container& cont = Container())
      : comp(cmp), c(cont) {
         make_heap(c.begin(),c.end(),comp);
      }
      void push(const value_type& x) {
         c.push_back(x);
         push_heap(c.begin(),c.end(),comp);
      }
      void pop() {
         pop_heap(c.begin(),c.end(),comp);
         c.pop_back();
      }
      bool               empty() const { return c.empty(); }
      size_type          size() const  { return c.size(); }
      const value_type&  top() const   { return c.front(); }
      ...
   };
}
```

Эти алгоритмы описаны в разделе 11.9.4.

Отметим, что в отличие от других контейнерных адаптеров, в очереди с приоритетами не определены операции сравнения. Подробное описание членов класса и операций приведено в разделе 12.4.

12.4. Подробное описание контейнерных адаптеров

Следующие подразделы содержат подробное описание членов и операций, определенных в контейнерных адаптерах `stack<>`, `queue<>` и `priority_queue<>`.

12.4.1. Определения типов

contadapt`::value_type`

- Тип элементов.
- Эквивалентно *container*`::value_type`.

contadapt`::reference`

- Тип ссылок на элементы.
- Эквивалентно *container*`::reference`.
- Доступно начиная со стандарта C++11.

contadapt`::const_reference`

- Тип ссылок на элементы, доступные только для чтения.
- Эквивалентно *container*`::const_reference`.
- Доступно начиная со стандарта C++11.

contadapt`::size_type`

- Целочисленный тип без знака для размерных значений.
- Эквивалентно *container*`::size_type`.

contadapt`::container_type`

- Тип контейнера.

12.4.2. Конструкторы

contadapt`::`*contadapt* `()`

- Конструктор по умолчанию.
- Создает пустой стек или очередь (с приоритетами).

`explicit` *contadapt*`::`*contadapt* `(const` *Container*`&` *cont*`)`
`explicit` *contadapt*`::`*contadapt* `(`*Container*`&&` *cont*`)`

- Создает стек или очередь, инициализированную элементами контейнера *cont*, который должен быть объектом класса контейнерного адаптера.
- В первой форме все элементы контейнера *cont* копируются.
- Во второй форме все элементы контейнера *cont* перемещаются, если передаваемый контейнер поддерживает семантику перемещения, в противном случае элементы копируются (доступно начиная со стандарта C++11).
- Обе формы не поддерживаются классом `priority_queue<>`.

По стандарту C++11 все конструкторы допускают передачу в качестве дополнительного аргумента механизма распределения памяти, используемого для размещения внутреннего контейнера.

12.4.3. Вспомогательные конструкторы для очередей с приоритетами

`explicit` *priority_queue*`::priority_queue (const CompFunc&` *op*`)`

- Создает пустую очередь с приоритетами с предикатом *op*, используемым как критерий сортировки.
- Примеры, демонстрирующие передачу критерия сортировки в качестве аргумента конструктора, приведены в разделах 7.7.5 и 7.8.6.

priority_queue`::priority_queue (const CompFunc&` *op* `const Container&` *cont*`)`

- Создает очередь с приоритетами, которая инициализируется элементами контейнера *cont* и использует предикат *op* в качестве критерия сортировки.
- Все элементы контейнера *cont* копируются.

priority_queue`::priority_queue (InputIterator` *beg*`, InputIterator` *end*`)`

- Создает очередь с приоритетами, которая инициализируется всеми элементами диапазона [*beg*,*end*).
- Эта функция является шаблонным членом (см. раздел 3.2), поэтому элементы диапазона-источника могут иметь любой тип, который может быть преобразован в тип элемента контейнера.

```
priority_queue::priority_queue (InputIterator beg,
                                InputIterator end,
                                const CompFunc& op)
```

- Создает очередь с приоритетами, которая инициализируется всеми элементами диапазона [*beg*,*end*) и использует предикат *op* как критерий сортировки.
- Эта функция является шаблонным членом (см. раздел 3.2), поэтому элементы диапазона-источника могут иметь любой тип, который может быть преобразован в тип элемента контейнера.
- Примеры, демонстрирующие передачу критерия сортировки в качестве аргумента конструктора, приведены в разделах 7.7.5 и 7.8.6.

```
priority_queue::priority_queue (InputIterator beg, InputIterator end,
                                const CompFunc& op, const Container& cont)
```

- Создает очередь с приоритетами, которая инициализируется всеми элементами контейнера *cont* и диапазона [*beg*,*end*) и использует предикат *op* как критерий сортировки.
- Эта функция является шаблонным членом (см. раздел 3.2), поэтому элементы диапазона-источника могут иметь любой тип, который может быть преобразован в тип элемента контейнера.

По стандарту C++11 все конструкторы допускают передачу в качестве дополнительного аргумента механизма распределения памяти, используемого для размещения внутреннего контейнера.

12.4.4. Операции

`bool` *contadapt*`::empty () const`

- Проверяет, пуст ли контейнерный адаптер (т.е. не содержит ни одного элемента).
- Эквивалентно *contadapt*`::size()==0`, но может работать быстрее.

`size_type` *contadapt*`::size () const`

- Возвращает текущее количество элементов.
- Для проверки, является ли контейнерный адаптер пустым (т.е. не содержит ни одного элемента), следует использовать функцию-член `empty()`, потому что она может работать быстрее.

`void` *contadapt*`::push (const` *value_type*`& elem)`
`void` *contadapt*`::push (`*value_type*`&& elem)`

- Первая форма вставляет копию аргумента *elem*.
- Первая форма перемещает аргумент *elem*, если поддерживается семантика перемещения, в противном случае аргумент *elem* копируется (начиная со стандарта C++11).

`void` *contadapt*`::emplace (`*args*`)`

- Вставляет новый элемент, инициализированный списком аргументов *args*.
- Доступно начиная со стандарта C++11.

reference contadapt`::top ()`
const_reference contadapt`::top () const`
reference contadapt`::front ()`
const_reference contadapt`::front () const`

- Все формы возвращают очередной элемент.
  - Для стека предусмотрены обе формы функции-члена `top()`, возвращающие элемент, вставленный последним.
  - Для очереди предусмотрены обе формы функции-члена `front()`, возвращающие элемент, вставленный первым.
  - Для очереди с приоритетами предусмотрена только вторая форма функции-члена `top()`, возвращающая элемент с максимальным значением. Если максимальное значение имеют несколько элементов, стандарт не определяет, какой именно элемент должен быть возвращен.
- Вызывающая сторона должна гарантировать, что контейнерный адаптер содержит хотя бы один элемент (`size()>0`), в противном случае последствия непредсказуемы.
- Форма, возвращающая неконстантную ссылку, позволяет модифицировать следующий доступный элемент, если он содержится в стеке или очереди. Насколько это соответствует хорошему стилю программирования, решать вам.
- До принятия стандарта C++11 типом возвращаемого значения был (`const`) `value_type&`, что обычно одно и то же.

`void` *contadapt*`::pop ()`

- Удаляет следующий элемент из контейнерного адаптера.
  - Для стека следующим является элемент, вставленный последним.
  - Для очереди следующим является элемент, вставленный первым.
  - Для очереди с приоритетами следующим является элемент, имеющий максимальное значение. Если максимальных элементов несколько, стандарт не определяет, какой из них следует удалить.
- Эта функция ничего не возвращает. Для обработки следующего элемента сначала необходимо вызвать функцию-член `top()` или `front()`.
- Вызывающая сторона должна гарантировать, что контейнерный адаптер содержит хотя бы один элемент (`size()>0`), в противном случае последствия непредсказуемы.

reference queue`::back ()`
const_reference queue`::back () const`

- Обе формы возвращают последний элемент очереди. Последним считается элемент, который был вставлен после всех остальных элементов очереди.
- Вызывающая сторона должна гарантировать, что контейнерный адаптер содержит хотя бы один элемент (`size()>0`), в противном случае последствия непредсказуемы.
- Первая форма для неконстантных очередей возвращает ссылку. Следовательно, можно модифицировать последний элемент, пока он пребывает в очереди. Насколько это соответствует хорошему стилю программирования, решать вам.
- До принятия стандарта C++11 типом возвращаемого значения был (`const`) `value_type&`, что обычно одно и то же.
- Предусмотрена только для класса `queue<>`.

`bool` *comparison* `(const` *contadapt*`& stack1, const` *contadapt*`& stack2)`

- Возвращает результат сравнения двух стеков или очередей одного и того же типа.
- Оператор *comparison* может быть одной из следующих:

  `==` и `!=`
  `<`, `>`, `<=` и `>=`

- Два стека или очереди одинаковы, если они имеют одинаковое количество элементов, расположенных в одинаковом порядке (все сравнения двух соответствующих элементов должны возвращать значение `true`).
- Для проверки того, что один из стеков или одна из очередей меньше другого стека или очереди, контейнерные адаптеры сравниваются лексикографически (см. раздел 11.5.4).
- Не предусмотрена для класса `priority_queue<>`.

`void` *contadapt*`::swap (`*contadapt*`& c)`
`void swap (`*contadapt*`& c1, `*contadapt*`& c2)`

- Обменивает содержимое объекта `*this` с содержимым объекта *c*, а также содержимое объектов *c1* и *c2* соответственно. Для очередей с приоритетами меняет местами также и критерии сортировки.
- Вызывает функцию-член `swap()` соответствующего контейнера (см. раздел 8.4).
- Доступна начиная со стандарта C++11.

12.5. Битовые множества

Битовые множества моделируют массивы фиксированного размера, состоящие из битов или булевых значений. Они полезны для управления наборами флагов, когда переменные могут представлять любую комбинацию флагов. Программы на языке C и старые программы на языке C++ для представления массивов битов обычно использовали тип `long` и манипулировали им с помощью побитовых операций, таких как `&`, `|` и `~`. Класс `bitset` имеет преимущество над типом `long`. Оно заключается в том, что битовые множества могут содержать произвольное количество битов и предусматривают дополнительные операции. Например, можно присваивать отдельные биты, а также читать и записывать битовые множества как последовательности, состоящие из `0` и `1`.

Отметим, что количество битов в битовом множестве изменить нельзя. Количество битов задается шаблонным параметром. Если нужен контейнер с переменным количеством битов или булевых значений, следует использовать класс `vector<bool>` (описанный в разделе 7.3.6).

Класс `bitset` определен в заголовочном файле `<bitset>`.

```
#include <bitset>
```

В заголовочном файле `<bitset>` класс `bitset` определен как шаблонный класс, в котором количество битов задается шаблонным параметром.

```
namespace std {
   template <size_t Bits>
   class bitset;
}
```

В этом случае шаблонный параметр представляет собой не тип, а целое число без знака (см. раздел 3.2).

Шаблоны с разными шаблонными аргументами считаются разными типами. Сравнивать и комбинировать можно лишь битовые множества, содержащие одинаковое количество битов.

Новшества стандарта C++11

В стандарте C++98 были определены практически все свойства битовых множеств. Перечислим самые важные дополнения, появившиеся в стандарте C++11.

- Битовые множества теперь можно инициализировать строковыми литералами (см. раздел 12.5.1).
- Преобразования в числовые значения (и наоборот) теперь поддерживают работу с типом `unsigned long long`. Для этого была предложена функция `to_ullong()` (см. раздел 12.5.1).
- Преобразования в строки (и наоборот) теперь позволяют задавать символ, интерпретируемый как установленный и сброшенный бит.
- Появилась функция-член `all()`, проверяющая, все ли биты установлены.
- Для использования битовых множеств в неупорядоченных контейнерах предусмотрена хеш-функция по умолчанию (см. раздел 7.9.2).

12.5.1. Примеры использования битовых множеств

Использование битовых множеств в качестве наборов флагов

Первый пример демонстрирует, как использовать битовые множества для управления наборами флагов. Каждый флаг имеет значение, определенное перечислением. Значение перечислимого типа используется как позиция в битовом множестве. В частности, биты представляют цвета. Таким образом, каждое значение перечисления определяет один цвет. Используя битовые множества, можно управлять переменными, которые могут содержать любую комбинацию цветов.

```
// contadapt/bitset1.cpp

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

int main()
{
   // тип перечисления для битов
   // - каждый бит представляет цвет
   enum Color { red, yellow, green, blue, white, black, ...,
                numColors };

   // создаем битовое множество для всех битов/цветов
   bitset<numColors> usedColors;

   // устанавливаем биты для двух цветов
   usedColors.set(red);
   usedColors.set(blue);

   // выводим на печать данные битового множества
   cout << "bitfield of used colors: " << usedColors << endl;
   cout << "number of used colors: " << usedColors.count() << endl;
   cout << "bitfield of unused colors: " << ~usedColors << endl;

   // если использован хотя бы один цвет
   if (usedColors.any()) {
      // цикл по всем цветам
      for (int c = 0; c < numColors; ++c) {
          // если использован текущий цвет
          if (usedColors[(Color)c]) {
           ...
          }
      }
   }
}
```

Использование битовых множеств для ввода и вывода битовых представлений

Полезным свойством битовых множеств является возможность преобразовывать целочисленные значения в последовательность битов, и наоборот. Это делается с помощью создания временного битового множества.

```
// contadapt/bitset2.cpp

#include <bitset>
#include <iostream>
#include <string>
#include <limits>
using namespace std;

int main()
{
   // выводим несколько чисел в битовом представлении
   cout << "267 as binary short: "
        << bitset<numeric_limits<unsigned short>::digits>(267)
        << endl;
   cout << "267 as binary long: "
        << bitset<numeric_limits<unsigned long>::digits>(267)
        << endl;
   cout << "10000000 with 24 bits: "
        << bitset<24>(1e7) << endl;

   // записываем битовое представление в строку
   string s = bitset<42>(12345678).to_string();
   cout << "12345678 with 42 bits: " << s << endl;

   // преобразовываем бинарное представление в целое число
   cout << "\"1000101011\" as number: "
        << bitset<100>("1000101011").to_ullong() << endl;
}
```

В зависимости от количества битов в типах `short` и `long long` программа может выдать следующий результат:

```
267 as binary short: 0000000100001011
267 as binary long: 00000000000000000000000100001011
10000000 with 24 bits: 100110001001011010000000
12345678 with 42 bits: 000000000000000000101111000110000101001110
"1000101011" as number: 555
```

В этом примере следующее выражение преобразовывает число `267` в битовое множество с количество битов, заданным числом типа `unsigned short` (см. раздел 5.3, посвященный числовым пределам):

```
bitset<numeric_limits<unsigned short>::digits>(267)
```

Оператор вывода для класса `bitset` выводит биты как последовательность символов `0` и `1`.

Битовые множества можно вывести непосредственно или использовать их значение как строку.

```
string s = bitset<42>(12345678).to_string();
```

До принятия стандарта C++11 здесь приходилось писать

```
string s = bitset<42>(12345678).to_string<char,char_traits<char>,
                                          allocator<char> >();
```

потому что `to_string()` — это шаблонная функция-член, а для шаблонных аргументов функций значения по умолчанию предусмотрены не были.

Аналогично следующее выражение преобразовывает последовательность бинарных символов в битовое множество, для которого функция-член `to_ullong()` выдает целое число:

```
bitset<100>("1000101011")
```

Отметим, что количество битов в битовом множестве не должно превышать `sizeof(unsigned long long)*8`. Причина заключается в том, что если битовое множество невозможно представить в виде числа типа `unsigned long long`, возникает исключение[2].

До принятия стандарта C++11 начальное значение необходимо было преобразовывать в тип `string` явным образом:

```
bitset<100>(string("1000101011"))
```

12.5.2. Подробное описание класса `bitset`

Из-за ограниченного объема книги подробное описание членов класса `bitset<>` размещено на веб-сайте, посвященном книге, по адресу `http://www.cppstdlib.com`.

[2] До принятия стандарта C++11 тип `unsigned long long` не был предусмотрен, поэтому здесь можно было вызывать только функцию `to_ulong()`. Эту функцию все еще можно вызывать, если количество битов меньше `sizeof(unsigned long)`.

Глава 13
Строки

Глава посвящена строковым типам из стандартной библиотеки C++. В ней описываются базовый шаблонный класс `basic_string<>` и его стандартные специализации `string`, `wstring`, `u16string` и `u32string`.

Строки могут быть источником недоразумений, потому что не всегда ясно, что именно обозначает слово "*строка*". Идет ли речь об обычном массиве символов типа `char*` (с квалификатором `const` или без него)? Имеется ли в виду экземпляр класса `string<>`? Или это общее название для объектов, представляющих собой разновидности строк? Термин "строка" в этой главе означает объект одного из строковых типов в стандартной библиотеке C++: `string`, `wstring`, `u16string` или `u32string`. Для "обычных строк" типа `char*` или `const char*` используется термин "С-строка".

В стандарте C++98 тип строковых литералов (таких как `"hello"`) был изменен на `const char*`. Однако для обеспечения обратной совместимости для них предусмотрено явное, но нежелательное преобразование в тип `char*`. Строго говоря, исходным типом литерала `"hello"` является `const char[6]`. Но этот тип автоматически приводится (*decays*) к `const char*`, поэтому практически всегда в объявлениях используется тип `const char*`. Тем не менее при работе с шаблонами разница между этими типами может оказаться важной, потому что для шаблонных ссылочных параметров приведение не выполняется, за исключением типа свойств `std::decay()` (см. раздел 5.4.2).

Изменения, внесенные в стандарте C++11

В стандарте C++98 описаны почти все строковые классы. Перечислим наиболее важные свойства, включенные в стандарт C++11.

- В строковые классы включены функции-члены `front()` и `back()` для доступа к первому или последнему элементу (см. раздел 13.2.6) и `shrink_to_fit()` для уменьшения выделенной памяти (см. раздел 13.2.5).
- В строковые классы включены удобные функции для преобразования строк в числа, и наоборот (см. раздел 13.2.13).
- Функции-члены `data()` и `c_str()` больше не делают некорректными ссылки, итераторы и указатели на строки (см. раздел 13.2.6).
- Строки стали поддерживать семантику перемещения (см. раздел 13.2.9) и списки инициализации (см. раздел 13.2.8).
- Кроме классов `string` и `wstring`, стандарт предусматривает две другие специализации класса `basic_string<>` — `u16string` и `u32string` (см. раздел 13.2.1).
- Строки стали неявно требовать наличия символа конца строки (`'\0'` для класса `string`), потому что для строки `s` выражение `s[s.length()]` всегда является

корректным, а функция `s.data()` возвращает символы, включая замыкающий символ конца строки (см. раздел 13.1.2).

- Реализации строковых классов с подсчетом ссылок больше не поддерживаются (см. раздел 13.2.16).

13.1. Предназначение строковых классов

Строковые классы стандартной библиотеки C++ позволяют работать со строками, как с обычными типами, не вызывающими у пользователей никаких затруднений. Таким образом, строки можно копировать, присваивать и сравнивать как элементарные типы, не беспокоясь о том, достаточно ли памяти или как долго останется корректной внутренняя память. Можно просто использовать операторы, такие как присваивания =, сравнения == и конкатенации +. Короче говоря, строковые типы стандартной библиотеки C++ разработаны так, чтобы не отличаться от элементарных типов и не вызывать никаких проблем (по крайней мере, теоретически). В настоящее время обработка данных во многом сводится к обработке строк, поэтому строки C++ стали важным шагом для программистов, ранее работавших на языках C, Fortran и подобных языках, в которых обработка строк связана с трудностями.

В следующих разделах приведены два примера, демонстрирующих возможности и использование строковых классов.

13.1.1. Первый пример: извлечение имени временного файла

В первом примере для создания имен временных файлов используются аргументы командной строки. Например, если запустить программу с помощью команды

```
string1 prog.dat mydir hello. oops.tmp end.dat
```

будет получен следующий вывод:

```
prog.dat => prog.tmp
mydir => mydir.tmp
hello. => hello.tmp
oops.tmp => oops.xxx
end.dat => end.tmp
```

Обычно временный файл имеет расширение `.tmp`, а имя временного файла с расширением `.tmp`, передаваемое в качестве аргумента, имеет расширение `.xxx`.

```
// string/string1.cpp

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

int main (int argc, char* argv[])
{
```

```
    string filename, basename, extname, tmpname;
    const string suffix("tmp");

    // для каждого аргумента командной строки
    // (который является обычной С-строкой)
    for (int i=1; i<argc; ++i) {
       // обрабатываем аргумент, как имя файла
       filename = argv[i];

      // ищем точку в имени файла
      string::size_type idx = filename.find('.');
      if (idx == string::npos) {
          // имя файла не содержит точки
          tmpname = filename + '.' + suffix;
      }
      else {
          // разделяем имя файла на основное имя и расширение
          // - основное имя состоит из всех символов, предшествующих точке
          // - расширение состоит из всех символов после точки
          basename = filename.substr(0, idx);
          extname = filename.substr(idx+1);
          if (extname.empty()) {
              // содержит точку, но не имеет расширения: добавляем tmp
              tmpname = filename;
              tmpname += suffix;
          }
          else if (extname == suffix) {
              // заменяем расширение tmp на xxx
              tmpname = filename;
              tmpname.replace (idx+1, extname.size(), "xxx");
          }
          else {
               // заменяем любое расширение на tmp
               tmpname = filename;
               tmpname.replace (idx+1, string::npos, suffix);
          }
       }
       // выводим имена исходного и временного файлов
       cout << filename << " => " << tmpname << endl;
    }
}
```

Во-первых, в программу включается заголовочный файл для стандартных строковых классов:

```
#include <string>
```

Как обычно, эти классы объявлены в пространстве имен `std`.

Следующее объявление создает четыре строковых переменных:

```
string filename, basename, extname, tmpname;
```

Так как здесь не передается ни один аргумент, для инициализации строк вызывается конструктор класса `string` по умолчанию, который выполняет инициализацию пустой строкой.

Следующее объявление создает константную строку `suffix`, которая используется в программе как обычный суффикс для имен временных файлов:

```
const string suffix("tmp");
```

Эта строка инициализируется обычной С-строкой, поэтому имеет значение `tmp`. Отметим, что С-строки можно комбинировать с объектами класса `string` почти во всех ситуациях, в которых можно комбинировать два объекта класса `string`. В частности, во всей программе каждое появление строки `suffix` можно непосредственно заменить С-строкой `"tmp"`.

На каждой итерации цикла `for` следующее выражение присваивает новое значение строковой переменной `filename`:

```
filename = argv[i];
```

В этом случае новое значение является обычной С-строкой. Однако на ее месте мог бы быть любой другой объект класса `string` или отдельный символ типа `char`.

Следующий код выполняет поиск первого[1] вхождения точки в строке `filename`:

```
string::size_type idx = filename.find('.');
```

Функция `find()` — одна из нескольких функций, предназначенных для поиска в строках. Возможен также поиск в обратном направлении, поиск подстроки, поиск части строки или поиск нескольких символов одновременно. Все эти функции поиска возвращают индекс первой совпадающей позиции. Да, возвращаемое значение представляет собой целое число, а не итератор. Обычный интерфейс для строк не основан на принципах библиотеки STL. Тем не менее строки до некоторой степени поддерживают концепцию итераторов (см. раздел 13.2.14). Тип возвращаемого значения у всех функций поиска — `string::size_type` — представляет собой целочисленный тип без знака, определенный в строковом классе[2]. Как обычно, индекс первого символа равен `0`. Индекс последнего символа равен "*количество символов* `-1`".

Если поиск завершился неудачей, в качестве признака этого события возвращается специальное значение `npos`, определенное в строковом классе. Таким образом, следующий оператор проверяет, успешно ли завершился поиск:

```
if (idx == string::npos)
```

Тип и значение `npos` часто служат причинами путаницы при работе со строками. Будьте очень внимательными и всегда используйте тип `string::size_type`, а *не* `int` или `unsigned` в качестве типа возвращаемого значения, когда хотите проверить значение, возвращаемое функцией поиска. В противном случае сравнение с `string::npos` может не сработать. Подробности изложены в разделе 13.2.12.

---

[1] К сожалению, из-за этого данная программа считает расширением (и соответственно заменяет) все, находящееся после *первой* точки, так что расширением файла `aaa.bbb.ccc` программа считает `bbb.ccc`, а не `ccc`. — *Примеч. консульт.*

[2] В частности, тип `size_type` для строк зависит от модели памяти, используемой в строковом классе. Подробности изложены в разделе 13.3.13.

Если поиск точки в нашем примере завершился неудачей, то имя файла не имеет расширения. В этом случае имя временного файла представляет собой конкатенацию имени исходного файла, точки и ранее определенного расширения для временных файлов:

```
tmpname = filename + '.' + suffix;
```

Таким образом, для конкатенации двух строк можно просто использовать оператор +. С его помощью можно также выполнять конкатенацию строк с обычными С-строками и отдельными символами.

Если точка найдена, выполняется раздел `else`. Здесь индекс точки используется для разделения имени файла на основное имя и расширение. Эту задачу выполняет функция-член `substr()`:

```
basename = filename.substr(0, idx);
extname = filename.substr(idx+1);
```

Первый параметр функции `substr()` — это начальный индекс. Необязательный второй аргумент — это количество символов, а не индекс конца. Если второй аргумент не используется, то все остальные символы возвращаются как подстрока. Во всех ситуациях, в которых индекс и длина используются как аргументы, строки подчиняются двум правилам.

1. Аргумент, определяющий **индекс**, должен иметь корректное значение. Это значение должно быть меньше, чем количество символов в строке (как обычно, индекс первого символа равен 0). Кроме того, индекс позиции, следующей за последним символом, задает конец строки.
2. В большинстве случаев использование индекса, превышающего количество символов, вызывает исключение `out_of_range`. Однако все функции, выполняющие поиск символа или позиции, допускают любой индекс. Если индекс превышает количество символов, эти функции просто возвращают значение `string::npos` ("не найден").
3. Аргумент, задающий **количество символов**, может иметь любое значение. Если размер больше количества оставшихся символов, используются все оставшиеся символы. В частности, значение `string::npos` всегда используется как синоним "все оставшиеся символы".

Таким образом, если точка не найдена, следующее выражение генерирует исключение:

```
filename.substr(filename.find('.'))
```

Однако выражение

```
filename.substr(0, filename.find('.'))
```

не генерирует исключение. Если точка не найдена, результатом является все имя файла.

Даже если точка будет найдена, расширение, возвращаемое функцией `substr()`, может оказаться пустым, потому что после точки символов больше нет. Это обстоятельство проверяется выражением

```
if (extname.empty())
```

Если это условие возвращает значение `true`, генерируемое имя временного файла становится обычным именем файла, к которому добавляется обычное расширение:

```
tmpname = filename;
tmpname += suffix;
```

Здесь для добавления расширения добавляется оператор +=.

Имя файла может уже иметь расширение временного файла. Для проверки этого факта используется оператор ==, сравнивающий две строки:

```
if (extname == suffix)
```

Если это сравнение возвращает значение `true`, то обычное расширение временных файлов заменяется расширением `xxx`:

```
tmpname = filename;
tmpname.replace (idx+1, extname.size(), "xxx");
```

где функция

```
extname.size()
```

возвращает количество символов в строке `extname`. Вместо функции `size()` можно было бы использовать функцию `length()`, выполняющую ту же задачу. Итак, обе функции, `size()` и `length()`, возвращают количество символов. В частности, функция `size()` никак не влияет на память, которую занимает строка[3].

Далее, после проверки всех специальных условий, выполняется обычная обработка. Программа заменяет все расширение обычным расширением временных файлов:

```
tmpname = filename;
tmpname.replace (idx+1, string::npos, suffix);
```

где значение `string::npos` используется как синоним "все оставшиеся символы". Таким образом, все оставшиеся символы после точки заменяются строкой `suffix`. Эта замена могла бы быть выполнена также, если имя файла содержит точку, но не имеет расширения. В этом случае "пустота" была бы заменена строкой `suffix`.

Оператор, записывающий имя исходного файла и сгенерированное имя временного файла, показывает, как можно вывести строки с помощью обычных операторов вывода в потоки (сюрприз!):

```
cout << filename << " => " << tmpname << endl;
```

13.1.2. Второй пример: извлечение слов и вывод их в обратном порядке

Второй пример извлекает целое слово из стандартного потока ввода и выводит символы каждого слова в обратном порядке. Слова разделяются обычными разделителями (символом перехода на новую строку, пробелом и символом табуляции), а также запятыми, точками и точками с запятой:

```
// string/string2.cpp

#include <iostream>
```

[3] В данном случае две функции-члена делают одно и то же на основе двух разных подходов: функция `length()` возвращает длину строки, аналогично функции `strlen()` для обычных С-строк, в то время как функция `size()` — это обычная функция-член, возвращающая количество элементов в соответствии с принципами библиотеки STL.

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

int main (int argc, char** argv)
{
   const string delims(" \t,.;");
   string line;

   // для каждой считанной строки
   while (getline(cin,line)) {
      string::size_type begIdx, endIdx;

      // поиск начинается с первого слова
      begIdx = line.find_first_not_of(delims);

      // ищем начало первого слова
      while (begIdx != string::npos) {
           // ищем конец текущего слова
           endIdx = line.find_first_of (delims, begIdx);
           if (endIdx == string::npos) {
              // конец слова - это конец строки
              endIdx = line.length();
           }

           // выводим символы в обратном порядке
           for (int i=endIdx-1; i>=static_cast<int>(begIdx); --i) {
                cout << line[i];
           }
           cout << ' ';


           // ищем начало следующего слова
           begIdx = line.find_first_not_of (delims, endIdx);
       }
       cout << endl;
   }
}
```

В этой программе все символы, используемые как разделители, определены в виде специальной строковой константы:

```
const string delims(" \t,.;");
```

Символ перехода на новую строку также используется как разделитель. Однако для него не требуется особая обработка, потому что программа читает строку за строкой.

Внешний цикл работает, пока строка из потока не будет прочитана в строку `line`:

```
string line;
while (getline(cin,line)) {
   ...
}
```

Функция `getline()` — это специальная функция, предназначенная для считывания содержимого входного потока в строку. Она считывает все символы до символа новой

строки. Сам символ-разделитель извлекается из потока, но не добавляется в строку. Передавая специальный символ разделения строк в качестве необязательного третьего аргумента, можно использовать функцию `getline()` для считывания лексем, разделенных специальным разделителем.

Во внешнем цикле выполняется поиск и вывод на печать отдельных слов. Первый оператор ищет начало первого слова:

```
begIdx = line.find_first_not_of(delims);
```

Функция `find_first_not_of()` возвращает первый индекс символа, не являющегося частью строки, передаваемой в виде аргумента. Таким образом, эта функция возвращает позицию первого символа, который не входит в набор разделителей, заданных в строке `delims`. Если совпадение не найдено, возвращается значение `string::npos`. Это типично для функций поиска.

Внутренний цикл повторяется, пока не будет найдено начало слова.

```
while (begIdx != string::npos) {
   ...
}
```

Первый оператор внутреннего цикла ищет конец текущего слова.

```
endIdx = line.find_first_of (delims, begIdx);
```

Функция `find_first_of()` ищет первое появление одного из символов, переданных как первый аргумент. В данном случае необязательный второй аргумент используется для того, чтобы задать начало поиска в строке. Следовательно, функция ищет первый разделитель после начала слова. Если такого символа нет, используется символ конца строки:

```
if (endIdx == string::npos) {
    endIdx = line.length();
}
```

Здесь использована функция `length()`, делающая то же самое, что и функция `size()`, т.е. возвращающая количество символов.

В следующем операторе все символы слова выводятся на печать в обратном порядке:

```
for (int i=endIdx-1; i>=static_cast<int>(begIdx); --i) {
     cout << line[i];
}
```

Доступ к отдельному символу в строке обеспечивается с помощью оператора `[]`. Отметим, что этот оператор *не проверяет* корректность индекса в строке. Таким образом, программист должен самостоятельно гарантировать, что индекс является корректным, как это сделано в программе. Более безопасным способом доступа к символу является использование функции-члена `at()`. Однако проверка корректности индекса занимает время, поэтому она не предусмотрена для обычной операции доступа к символам строки.

Отметим, что для оператора `[]` количество символов является корректным индексом, соответствующим символу конца строки. *Символ конца строки* инициализируется конструктором по умолчанию для символьного типа (`'\0'` для класса `string`).[4]

```
string s;
s[s.length()] // выдает '\0'
```

[4] До принятия стандарта C++11 для неконстантной версии оператора `[]` текущее количество символов было некорректным индексом и его использование приводило к неопределенному поведению.

Другая неприятность исходит от использования индекса строки. Если пропустить приведение переменной `begIdx` к типу `int`, то программа может войти в бесконечный цикл или закончится аварийно. Как и в первой программе, проблема заключается в том, что тип `string::size_type` является целочисленным типом без знака. Без приведения знаковое значение `i` автоматически преобразуется в беззнаковое, потому что оно сравнивается с беззнаковым значением. В этом случае, если текущее слово начинается с начала строки, приведенное далее выражение всегда возвращает `true`.

```
i>=begIdx
```

Дело в том, что переменная `begIdx` для слова в начале строки равна `0`, а любое беззнаковое значение больше или равно `0`. Так возникает бесконечный цикл, который приводит к краху из-за неверного обращения к памяти. По этой причине я не люблю концепцию `string::size_type` и `string::npos`. Более безопасный, но не идеальный способ решения этой проблемы изложен в разделе 13.2.12.

Последний оператор внутреннего цикла повторно инициализирует переменную `begIdx` началом следующего слова, если оно есть.

```
begIdx = line.find_first_not_of (delims, endIdx);
```

В отличие от первого вызова функции `find_first_not_of()` в нашем примере, в качестве начального индекса для поиска передается конец предыдущего слова. Если предыдущее слово было последним в строке, значение `endIdx` представляет собой индекс конца строки. Это означает лишь то, что поиск начинается с конца строки и функция возвращает значение `string::npos`.

Попробуем выполнить эту "полезную и важную" программу. Вот как выглядят ее входные данные[5]:

```
pots & pans
I saw a reed
deliver no pets
nametag on diaper
```

Результаты обработки входных данных программы приведены ниже.

```
stop & snap
I was a deer
reviled on step
gateman no repaid
```

13.2. Описание строковых классов

13.2.1. Строковые типы

Заголовочный файл

Все типы и функции для строк определены в заголовочном файле `<string>`.

```
#include <string>
```

Как обычно, все идентификаторы в нем определены в пространстве имен `std`.

[5] Две последние строки подсказаны Шоном Окифи (Sean Okeefe).

Шаблонный класс basic_string<>

В заголовочном файле <string> класс basic_string<> определен как базовый класс для всех строковых типов.

```
namespace std {
   template <typename charT,
             typename traits = char_traits<charT>,
             typename Allocator = allocator<charT> >
             class basic_string;
}
```

Этот класс параметризован символьным типом, свойствами символьного типа и моделью памяти.

- Первый параметр — это тип данных отдельного символа.
- Необязательный второй параметр — это класс свойств, содержащий все операции над символами строкового класса. Такой класс свойств определяет, как копировать или сравнивать символы (подробнее об этом речь пойдет в разделе 16.1.4). Если этот класс не определен, используется класс свойств, установленный по умолчанию для текущего символьного типа. В разделе 13.2.15 приведен пользовательский класс свойств, позволяющий обрабатывать строки независимо от регистра.
- Третий необязательный параметр определяет модель памяти, используемую строковым классом. Как обычно, по умолчанию используется модель памяти allocator (подробности изложены в разделе 4.6 и в главе 19).

Конкретные строковые типы

В стандартной библиотеке C++ содержится несколько специализаций базового класса basic_string<>.

- Класс string является предопределенной специализацией шаблона для символов типа char:

```
namespace std {
   typedef basic_string<char> string;
}
```

- Три остальных типа определены для строк, использующих более широкий набор символов, такой как Unicode или некоторые азиатские алфавиты (типы u16string и u32string появились вместе со стандартом C++11):

```
namespace std {
   typedef basic_string<wchar_t>  wstring;
   typedef basic_string<char16_t> u16string;
   typedef basic_string<char32_t> u32string;
}
```

Вопросы интернационализации освещены в главе 16.

В следующих разделах не проводятся различия между этими типами строк. Способы использования и проблемы, связанные с ними, не отличаются друг от друга, потому что все строковые классы имеют одинаковый интерфейс. Итак, слово "строка" означает

любой из строковых типов: `string`, `wstring`, `u16string` и `u32string`. В примерах, приведенных в книге, обычно используется тип `string`, потому что для разработчиков программного обеспечения естественной средой является Европа и англоязычная Америка.

Таблица 13.1. Операции над строками

Операция	Действие
конструкторы	Создают или копируют строку
деструктор	Уничтожает строку
`=, assign()`	Присваивает новое значение
`swap()`	Обмен двух строк
`+=, append(), push_back()`	Добавляет символы
`insert()`	Вставляет символы
`erase(), pop_back()`	Удаляет символы (`pop_back()`, начиная со стандарта C++11)
`clear()`	Удаляет все символы (делает строку пустой)
`resize()`	Изменяет количество символов (удаляет или добавляет символы в конец)
`replace()`	Заменяет символы
`+`	Выполняет конкатенацию строк
`==, !=, <, <=, >, >=, compare()`	Сравнивают строки
`empty()`	Проверяет, является ли строка пустой
`size(), length()`	Возвращают количество символов
`max_size()`	Возвращает максимально возможное количество символов
`capacity()`	Возвращает количество символов, которые можно хранить без повторного выделения памяти
`reserve()`	Резервирует память для определенного количества символов
`shrink_to_fit()`	Уменьшает размер памяти до текущего количества символов (начиная со стандарта C++11)
`[], at()`	Обеспечивают доступ к символу
`front(), back()`	Обеспечивают доступ к первому или последнему символу (начиная со стандарта C++11)
`>>, getline()`	Читают значение из потока
`<<`	Записывает значение в поток
`stoi(), stol(), stoll()`	Преобразовывают строку в целое число со знаком (начиная со стандарта C++11)
`stoul(), stoull()`	Преобразовывают строку в целое число без знака (начиная со стандарта C++11)
`stof(), stod(), stold()`	Преобразовывают строку в число с плавающей точкой (начиная со стандарта C++11)

Окончание табл. 13.1

Операция	Действие
`to_string(),to_wstring()`	Преобразовывают целое число или число с плавающей точкой в строку (начиная со стандарта C++11)
`copy()`	Копирует или записывает содержимое массива символов
`data(),c_str()`	Возвращают значение C-строки или массив символов
`substr()`	Возвращает заданную подстроку
Функции поиска	Выполняют поиск заданной подстроки или символа
`begin(),end()`	Обеспечивают поддержку обычных итераторов
`cbegin(),cend()`	Обеспечивают поддержку константных итераторов (начиная со стандарта C++11)
`rbegin(),rend()`	Обеспечивают поддержку обратных итераторов
`crbegin(),crend()`	Обеспечивают поддержку константных обратных итераторов (начиная со стандарта C++11)
`get_allocator()`	Возвращает механизм выделения памяти

13.2.2. Обзор операций

В табл. 13.1 приведены все операции, предусмотренные для строк.

Аргументы операций над строками

Существует много операций для манипулирования строками. В частности, операции, изменяющие строки, имеют несколько перегруженных версий, которые задают новое значение, с одним, двумя или тремя аргументами. Все эти операции используют схему аргументов, представленную в табл. 13.2.

Таблица 13.2. Схема аргументов операций над строками

Аргументы	Интерпретация
`const string &` *str*	Вся строка *str*
`const string &` *str*, `size_type` *idx*, `size_type` *num*	Не более *num* первых символов строки *str*, начиная с индекса *idx*
`const char*` *cstr*	Вся C-строка *cstr*
`const char*` *chars*, `size_type` *len*	*len* символов массива символов *chars*
`char` *c*	Символ *c*
`size_type` *num*, `char` *c*	*num* вхождений символа *c*
`const_iterator` *beg*, `const_iterator` *end*	Все символы диапазона [*beg*,*end*)
список инициализации	Все символы в списке инициализации (начиная со стандарта C++11)

Отметим, что только версия с одним аргументом `const char*` интерпретирует символ `'\0'` как специальный символ, завершающий строку. Во всех остальных случаях символ `'\0'` *не является* специальным.

```
std::string s1("nico");  // инициализирует s1 символами 'n' 'i' 'c' 'o'
std::string s2("nico",5);// инициализирует s2 символами 'n' 'i' 'c' 'o' '\0'
std::string s3(5,'\0');  // инициализирует s3 символами '\0' '\0' '\0' '\0' '\0'

s1.length()              // возвращает 4
s2.length()              // возвращает 5
s3.length()              // возвращает 5
```

Таким образом, в принципе строка может содержать любой символ. В частности, строка может содержать содержимое бинарного файла.

Передача нулевого указателя в качестве аргумента `cstr` приводит к неопределенному поведению.

В табл. 13.3 приведен обзор операций, использующих разные аргументы. Все операторы могут работать как с отдельными значениями только с объектами. Следовательно, для того чтобы присвоить, сравнить или добавить часть строки или С-строки, необходимо использовать функцию, имеющую соответствующее имя.

Таблица 13.3. Доступные операции со строковыми параметрами

	Полная строка	Часть строки	С-строка (char*)	Массив символов	Отдельный символ	пип символов	Диапазон итераторов	Список инициализации
Конструкторы	Да	Да	Да	Да	—	Да	Да	Да
=	Да	—	Да	—	Да	—	—	Да
`assign()`	Да	Да	Да	Да	—	Да	Да	Да
+=	Да	—	Да	—	Да	—	—	Да
`append()`	Да	Да	Да	Да	—	Да	Да	Да
`push_back()`	—	—	—	—	Да	—	—	—
`insert()` для индексов	Да	Да	Да	Да	—	Да	—	—
`insert()` для итераторов	—	—	—	—	Да	Да	Да	Да
`replace()` для индексов	Да	Да	Да	Да	Да	Да	—	—
`replace()` для итераторов	Да	—	Да	Да	—	Да	Да	Да
Функции поиска	Да	—	Да	Да	Да	—	—	—
+	Да	—	Да	—	Да	—	—	—
==, !=, <, <=, >, >=	Да	—	Да	—	—	—	—	—
`compare()`	Да	Да	Да	Да	—	—	—	—

Не предусмотренные операции

Строковые классы стандартной библиотеки C++ не решают все возможные задачи работы со строками. Фактически они не предлагают прямых решений для регулярных выражений и обработки текста. Однако для работы с регулярными выражениями в стандарте C++11 предназначена отдельная библиотека. (Эта тема будет обсуждаться в главе 14.) Для обработки текста (преобразования в верхний регистр, сравнений без учета регистров) в разделе 13.2.14 приведено несколько примеров.

13.2.3. Конструкторы и деструкторы

В табл. 13.4 перечислены все конструкторы и деструкторы для строк.

Строку невозможно инициализировать отдельным символом. Вместо этого следует использовать его адрес или дополнительное количество вхождений или формат списка инициализации (начиная со стандарта C++11):

```
std::string s('x');     // ОШИБКА
std::string s(1,'x');   // OK, создает строку из одного символа 'x'
std::string s({'x'});   // OK, то же самое (начиная со стандарта C++11)
```

Это значит, что можно автоматически преобразовать в `string` тип `const char*`, но не тип `char`.

Инициализация диапазоном, определяемая итераторами, описана в разделе 13.2.14.

Таблица 13.4. Конструкторы и деструкторы строк

Выражение	Действие
`string s`	Создает пустую строку *s*
`string s(`*str*`)`	Копирующий конструктор; создает строку, являющуюся копией существующей строки *str*
`string s(`*rvStr*`)`	Перемещающий конструктор; создает строку и перемещает в нее содержимое строки *rvStr* (строка *rvStr* должна иметь корректное состояние, а после перемещения ее значение не определено)
`string s(`*str*`,`*stridx*`)`	Создает строку *s*, инициализированную символами строки *str*, начинающимися с индекса *stridx*
`string s(`*str*`,`*stridx*`,`*strlen*`)`	Создает строку *s*, инициализированную не более чем *strlen* символами строки *str*, начинающихся с индекса *stridx*
`string s(`*cstr*`)`	Создает строку *s*, инициализированную С-строкой *cstr*
`string s(`*chars*`,`*charslen*`)`	Создает строку *s*, инициализированную *charslen* символами массива символов *chars*
`string s(`*num*`,`*c*`)`	Создает строку, имеющую *num* вхождения символа *c*
`string s(`*beg*`,`*end*`)`	Создает строку, инициализированную всеми символами диапазона [*beg*,*end*)
`string s(`*initlist*`)`	Создает строку, инициализированную всеми символами из списка инициализации (начиная со стандарта C++11)
`s.~string()`	Уничтожает все символы и освобождает память

13.2.4. Строки и С-строки

В стандарте языка C++ тип строковых литералов был изменен с `char*` на `const char*`. Однако для обеспечения обратной совместимости для них сохранено неявное, но нежелательное преобразование в тип `char*`. Поскольку строковые литералы не имеют типа `string`, между объектами класса `string` и обычными С-строками существует сильная связь: обычные С-строки можно использовать практически во всех ситуациях, в которых строки сочетаются с другими строковыми объектами (при сравнении, добавлении, вставке и т.д.). В частности, существует автоматическое преобразование из типа `const char*` в строки. Однако автоматического преобразования строки в С-строку *нет*. Это сделано по соображениям безопасности, чтобы предотвратить непреднамеренные преобразования типа, порождающие странные последствия (тип `char*` часто ведет себя странно) и неоднозначности. Например, в выражении, в котором комбинируются объект класса `string` и С-строка, были бы возможны как преобразования `string` в `char`, так и обратные[6]. Вместо этого имеется несколько способов создать и записать или скопировать объект класса `string` в С-строку. В частности, функция `c_str()` позволяет сгенерировать значение строки в виде С-строки как символьный массив, в котором последним символом является `'\0'`. Используя функцию `copy()`, можно копировать или записывать значение в существующую С-строку или символьный массив.

Отметим, что строки *не* придают специального смысла символу `'\0'`, который в обычных С-строках играет роль конца строки. Символ `'\0'` может быть частью строки, как и любой другой символ.

Кроме того, если вы используете нулевой указатель старого стиля (`NULL`), а не `nullptr` (см. раздел 3.1.1) или параметр типа `char*`, то в результате возникают странные последствия. Причина заключается в том, что `NULL` — это значение целочисленного типа, которое интерпретируется как `0`, или символ со значением `0`, если операция перегружена для отдельного целочисленного типа. Итак, всегда следует использовать указатели `nullptr` или `char*`.

Существуют три способа преобразовать содержимое строки в обычный массив символов или С-строку.

1. Функции **`data()`** и **`c_str()`** возвращают содержимое строки в виде массива символа. Массив содержит *символ конца строки* в позиции `[size()]`, поэтому для объектов класса `string` результатом является корректная С-строка, включающая символ `'\0'`.
2. Отметим, что до появления стандарта C++11 тип значения, возвращаемого функцией `data()`, *не был* корректной С-строкой, потому что добавление символа `'\0'` не гарантировалось.
3. Функция **`copy()`** копирует содержимое строки в массив символов, предоставленный вызывающей стороной. Символ `'\0'` не добавляется.
4. Кроме того, функции `data()` и `c_str()` возвращают массив, которым владеет строка. Следовательно, вызывающая сторона не должна модифицировать или освобождать ее память. Рассмотрим пример:

[6] Кроме того, при преобразовании строк в С-строки, которые по сути являются указателями, в игру вступала бы еще и арифметика указателей, что совершенно запутывало бы дело (например, как бы выполнялась проверка равенства строк — по их содержимому или как двух указателей?). — *Примеч. консульт.*

```
std::string s("12345");

atoi(s.c_str())              // преобразуем строку в целое число
f(s.data(),s.length())       // вызываем функцию, передавая ей массив
                             // символов и их количество
char buffer[100];
s.copy(buffer,100);          // копируем не более 100 символов строки s
                             // в buffer
s.copy(buffer,100,2);        // копируем не более 100 символов строки s
                             // в buffer, начиная с третьего символа
```

Строки обычно используются по всей программе, поэтому преобразовывать их в С-строки или массивы символов необходимо только непосредственно перед тем, как вам потребуется содержимое типа `char*`. Отметим, что тип значений, возвращаемых функциями `c_str()` и `data()`, является корректным только до следующего вызова неконстантной функции-члена для той же строки.

```
std::string s;
...
foo(s.c_str());        // строка s.c_str() корректна в рамках выражения

const char* p;
p = s.c_str();         // p ссылается на содержимое s как С-строки
foo(p);                // ОК (p остается корректным)
s += "ext";            // указатель p становится некорректным
foo(p);                // ОШИБКА: аргумент p более не корректен!
```

13.2.5. Размер и емкость

Для эффективного и корректного использования строк необходимо понимать, как связаны между собой размер и емкость. У строк есть три "размера".

1. Функции **`size()`** и **`length()`** — эквивалентные функции, возвращающие текущее количество символов в строке[7].
2. Функция-член **`empty()`** проверяет, равно ли количество символов нулю. Следовательно, она проверяет, является ли строка пустой. Для обеспечения высокого быстродействия следует использовать функцию `empty()`, а не `length()` или `size()`.
3. Функция **`max_size()`** возвращает максимальное количество символов, которое может содержать строка. Строка обычно содержит все символы в одном блоке памяти, поэтому могут существовать ограничения, связанные с компьютером. В противном случае это значение обычно равно максимальному значению типа индекса минус один. Единица вычитается по двум причинам: а) константа `npos` равна максимальному значению и б) реализация может автоматически добавлять `'\0'` в конец буфера, так что она просто возвращает этот буфер, если строка используется как С-строка (например, функцией `c_str()`). Если результатом операции является строка, длина которой больше `max_size()`, класс генерирует исключение `length_error`.

[7] В этом случае две функции-члена делают одно и то же, потому что функция `length()` возвращает длину строки, функция `strlen()` — длину С-строки, а функция `size()` — это обычная функция-член, возвращающая количество элемента контейнера в соответствии с концепцией библиотеки STL.

4. Функция **capacity()** возвращает количество символов, которое может содержать строка без повторного выделения памяти.

Достаточная емкость важна по двум причинам.

1. Повторное выделение памяти делает некорректными все ссылки, указатели и итераторы, ссылающиеся на символы строки.
2. Повторное выделение памяти требует времени.

Таким образом, емкость следует учитывать, если в программе используются указатели, ссылки или итераторы, ссылающиеся на строку или символы строки, или если определяющим является быстродействие.

Предотвратить повторное выделение памяти позволяет функция-член `reserve()`. Она резервирует определенную емкость до того, как она действительно потребуется, чтобы сохранить корректность ссылок.

```
std::string s;  // создаем пустую строку
s.reserve(80);  // резервируем память для 80 символов
```

Концепция емкости строк в принципе совпадает с концепцией емкости контейнеров (см. раздел 7.3.1). Однако существует одно важное отличие: в отличие от векторов, вызывая функцию `reserve()` для строк, можно уменьшать их емкость. Вызов функции `reserve()` с аргументом, который меньше, чем текущая емкость, является необязательным запросом на уменьшение емкости. Если аргумент меньше текущего количества символов, то вызов представляет собой необязательный запрос на уменьшение до размера строки. Таким образом, несмотря на то, что вы можете *захотеть* уменьшить емкость, нет гарантии, что это произойдет. По умолчанию значение функции `reserve()` для строки равно `0`. Итак, вызов функции `reserve()` без аргументов всегда представляет собой необязательный запрос на уменьшение до размера строки:

```
s.reserve(); // "Я хотел бы уменьшить емкость до текущего размера"
```

После принятия стандарта C++11 функция `shrink_to_fit()` действует точно так же:

```
s.shrink_to_fit(); // "Я хотел бы уменьшить емкость до текущего размера" (C++11)
```

Вызов для уменьшения емкости является необязательным, потому что достижение оптимального быстродействия зависит от реализации. Реализации строкового класса могут использовать разные подходы к проектированию с точки зрения скорости и использования памяти. Например, реализации могут увеличивать емкость большими порциями или никогда не уменьшать емкость.

В то же время стандарт регламентирует, что емкость можно уменьшать только при вызове функции `reserve()` или `shrink_to_fit()`. Это гарантирует, что указатели, ссылки и итераторы остаются корректными, даже если символы были удалены или изменены, при условии, что они ссылаются на символы, занимающие позицию перед обрабатываемыми символами.

13.2.6. Доступ к элементам

Строка позволяет читать и записывать символы, которые она содержит. Доступ к отдельному символу обеспечивают оператор индексирования `[]` и функция-член `at()`. Начиная со стандарта C++11 функции `front()` и `back()` обеспечивают доступ к первому и последнему символам соответственно.

Все эти операции возвращают ссылку на символ, находящийся в позиции с указанным индексом. Если строка является константной, то символ также считается константным. Как обычно, первый символ имеет индекс 0, а последний символ — `length()-1`. Однако имеется несколько отличий.

- Оператор `[]` *не* проверяет корректность индекса, передаваемого в качестве аргумента, а функция `at()` проверяет. Если функции `at()` передается некорректный индекс, она генерирует исключение `out_of_range`. Если оператор `[]` вызывается с некорректным индексом, то его поведение становится неопределенным. В результате может произойти недопустимое обращение к памяти, способное привести к ужасным последствиям или краху (крах в этом случае можно считать везением, потому что вы хотя бы будете знать, что произошло нечто неправильное).
- В общем случае позиция за последним элементом является корректной. Следовательно, текущее количество символов является корректным индексом. Оператор возвращает значение, генерируемое конструктором по умолчанию для символьного типа. Таким образом, для объектов типа `string` он возвращает `char '\0'`.[8]
- Функция `front()` эквивалентна оператору `[0]`. Это значит, что для пустой строки возвращается символ, представляющий конец строки (`'\0'` для `string`).
- Для функции `at()` текущее количество символов не является корректным индексом.
- При вызове для пустой строки функция `back()` приводит к непредсказуемым последствиям.

Вот некоторые примеры:

```
const std::string cs("nico");       // cs содержит: 'n' 'i' 'c' 'o'
std::string s("abcde");             // s содержит: 'a' 'b' 'c' 'd' 'e'
std::string t;                      // t не содержит символов (пустая)

s[2]                                // возвращает 'c' как char&
s.at(2)                             // возвращает 'c' как char&
s.front()                           // возвращает 'a' как char&
cs[2]                               // возвращает 'c' как const char&
cs.back()                           // возвращает 'o' как const char&
s[100]                              // ОШИБКА: непредсказуемые последствия

s.at(100)                           // генерирует исключение out_of_range
t.front()                           // возвращает '\0'
t.back()                            // ОШИБКА: неопределенное поведение

s[s.length()]                       // возвращает '\0' (неопределенное
                                    // поведение до принятия стандарта C++11)
cs[cs.length()]                     // возвращает '\0'
s.at(s.length())                    // генерирует исключение out_of_range
cs.at(cs.length())                  // генерирует исключение out_of_range
```

Для того чтобы изменить символ в строке, неконстантные версии оператора `[]` и функций `at()`, `front()` и `back()` возвращают ссылку на символ. Отметим, что при перераспределении памяти ссылка становится некорректной.

[8] До принятия стандарта C++11 для неконстантной версии оператора `[]` текущее количество символов было некорректным индексом. Его использование приводило к непредсказуемым последствиям.

```
std::string s("abcde");          // s содержит: 'a' 'b' 'c' 'd' 'e'

char& r = s[2];                  // ссылка на третий символ
char* p = &s[3];                 // указатель на четвертый символ

r = 'X';                         // OK, s содержит: 'a' 'b' 'X' 'd' 'e'
*p = 'Y';                        // OK, s содержит: 'a' 'b' 'X' 'Y' 'e'

s = "new long value";            // при повторном выделении памяти
                                 // r и p становятся некорректными

r = 'X';                         // ОШИБКА: непредсказуемые последствия
*p = 'Y';                        // ОШИБКА: непредсказуемые последствия
```

Для того чтобы избежать ошибок во время выполнения программы, следует зарезервировать достаточную емкость до того, как будут инициализированы ссылка `r` и указатель `p`.

Ссылки, указатели и итераторы, ссылающиеся на символы в строке, могут стать недействительными при выполнении следующих операций[9].

- При обмене значений с помощью функции `swap()`
- При чтении нового значения с помощью оператора `operator>>()` или функции `getline()`
- При вызове неконстантной функции-члена, за исключением оператора `[]` и функций `at()`, `begin()`, `end()`, `rbegin()` и `rend()`

Подробности, касающиеся строковых итераторов, изложены в разделе 13.2.14.

13.2.7. Сравнения

Для строк предусмотрены обычные операторы сравнения. Их операндами могут быть как строки, так и С-строки.

```
std::string s1, s2;
...
s1 == s2      // возвращает true, если s1 и s2 содержит одни и те же символы
s1 < "hello"  // проверяет, меньше ли s1 С-строки "hello"
```

Если строки сравниваются с помощью операторов <, <=, > или >=, то их символы сравниваются лексикографически в соответствии со свойствами текущего символа. Например, все следующие сравнения возвращают значение `true`:

```
std::string("aaaa") < std::string("bbbb")
std::string("aaaa") < std::string("abba")
std::string("aaaa") < std::string("aaaaaa")
```

Используя функции-члены `compare()`, можно сравнивать подстроки. Функции-члены `compare()` допускают описание строки несколькими аргументами, поэтому подстроку можно задавать по ее индексу и длине. Отметим, что функция `compare()` возвращает

[9] До принятия стандарта C++11 функции `data()` и `c_str()` также могли делать некорректными ссылки, итераторы и указатели на строки.

целое число, а не булево значение. Это число имеет следующий смысл: 0 означает "равно", меньше нуля — "меньше", а больше нуля — "больше". Например:

```
std::string s("abcd");
s.compare("abcd")        // возвращает 0
s.compare("dcba")        // возвращает значение < 0 (s меньше)
s.compare("ab")          // возвращает значение > 0 (s больше)
s.compare(s)             // возвращает 0 (s равно s)
s.compare(0,2,s,2,2)     // возвращает значение < 0 ("ab" меньше "cd")
s.compare(1,2,"bcx",2)   // возвращает 0 ("bc" равно "bc")
```

Для использования разных критериев сравнения можно определить свой собственный критерий и применить алгоритмы сравнения из библиотеки STL (см. раздел 13.2.14) или использовать специальные свойства символов, предусматривающие сравнение с учетом регистра. Однако строковый класс, имеющий специальный класс свойств, представляет собой отдельный тип данных; такие строки нельзя комбинировать с объектами типа `string`. Пример приведен в разделе 13.2.15.

Если программа предназначена для международного рынка, может возникнуть необходимость сравнивать строки с учетом локализации. Класс `locale` содержит оператор `()`, позволяющий легко решить эту задачу (см. раздел 16.3). Он использует аспект контекстного сравнения, предусмотренный для сравнения строк в соответствии с национальными стандартами. Подробности изложены в разделе 16.4.5.

13.2.8. Модифицирующие операции

Строки можно модифицировать с помощью различных функций-членов и операторов.

Присваивание

Чтобы присвоить строке новое значение, можно использовать оператор =. Присваиваемое значение может быть строкой, С-строкой или отдельным символом. Кроме того, с помощью функций-членов `assign()` можно выполнять присваивание строк в ситуациях, когда новое значение описывается несколькими аргументами. Например:

```
const std::string aString("othello");
std::string s;

s = aString;             // присваиваем "othello"
s = "two\nlines";        // присваиваем С-строку
s = ' ';                 // присваиваем отдельный символ

s.assign(aString);       // присваиваем "othello" (эквивалент оператора =)
s.assign(aString,1,3);   // присваиваем "the"
s.assign(aString,2,std::string::npos); // присваиваем "hello"

s.assign("two\nlines"); // присваиваем С-строку (эквивалент оператора =)
s.assign("nico",5);      // присваиваем массив символов: 'n' 'i' 'c' 'o' '\0'
s.assign(5,'x');         // присваиваем пять символов: 'x' 'x' 'x' 'x' 'x'
```

Можно также присвоить диапазон символов, заданный двумя итераторами. Подробности изложены в разделе 13.2.14.

Обмен значений

Как и во многих нетривиальных типах, в строковом типе предусмотрена специализация функции `swap()`, выполняющей обмен содержимого двух строк (глобальная функция `swap()` описана в разделе 5.5.2). Специализация функции `swap()` для строк обеспечивает константную сложность, поэтому ее следует использовать для обмена значениями строк, а также для присваивания строк в ситуации, когда присваиваемая строка после выполнения операции больше не нужна.

Очистка строк

Для удаления всех символов из строки существует несколько возможностей:

```
std::string s;

s = "";      // присваиваем пустую строку
s.clear();   // очищаем содержимое
s.erase();   // удаляем все символы
```

Вставка и удаление символов

Строковый класс содержит множество функций-членов для вставки, удаления, замены и стирания символов в строке. Для того чтобы добавить символы, можно использовать оператор += и функции `append()` и `push_back()`. Например:

```
const std::string aString("othello");
std::string s;

s += aString;             // добавляем "othello"
s += "two\nlines";        // добавляем С-строку
s += '\n';                // добавляем отдельный символ
s += { 'o', 'k' };        // добавляем список инициализации символов (C++11)
s.append(aString);        // добавляем "othello" (эквивалент оператора +=)
s.append(aString,1,3);    // добавляем "the"
s.append(aString,2,std::string::npos); // добавляем "hello"
s.append("two\nlines");   // добавляем С-строку (эквивалент оператора +=)
s.append("nico",5);       // добавляем массив символов: 'n' 'i' 'c' 'o' '\0'
s.append(5,'x');          // добавляем пять символов: 'x' 'x' 'x' 'x' 'x'
s.push_back('\n');        // добавляем отдельный символ (эквивалент оператора +=)
```

Оператор += добавляет значения, задаваемые одним аргументом, включая списки инициализации символов, начиная со стандарта C++11. Функция `append()` имеет перегруженные варианты для разных аргументов. Одна из версий функции `append()` позволяет добавить диапазон символов, заданный двумя итераторами (см. раздел 13.2.14). Функция-член `push_back()` предназначена для вставки символов в конец, так что алгоритмы STL могут добавлять символы в строку (см. раздел 9.4.2, в котором изложена информация об итераторах вставки в конец, и раздел 13.2.14, в котором приведен пример их использования для строк).

Аналогично функции `append()`, несколько вариантов функции-члена `insert()` позволяют вставлять символы. Эти функции получают индекс символа, после которого должны быть вставлены новые символы.

```
const std::string aString("age");
std::string s("p");
s.insert(1,aString);     // s: page
s.insert(1,"ersifl");    // s: persiflage
```

Отметим, что не существует варианта функции-члена `insert()`, который получал бы индекс и отдельный символ. Таким образом, следует передавать строку или дополнительное число.

```
s.insert(0,' ');    // ОШИБКА
s.insert(0," ");    // OK
```

Можно также попробовать выполнить оператор

```
s.insert(0,1,' '); // ОШИБКА: неоднозначность
```

Однако он приводит к неоднозначности, потому что функция `insert()` перегружена для следующих сигнатур.

```
insert (size_type idx, size_type num, charT c); // позиция задана индексом
insert (iterator pos, size_type num, charT c);  // позиция задана итератором
```

Для класса `string` тип `size_type` обычно определен как `unsigned`, а `iterator` часто определяется как `char*`. В этом случае первый аргумент `0` имеет два эквивалентных преобразования. Итак, чтобы добиться правильного результата, надо написать

```
s.insert((std::string::size_type)0,1,' '); // OK
```

Вторая интерпретация неоднозначности, описанной выше, — пример использования итераторов для вставки символов. Если требуется указать позицию для вставки с помощью итератора, существуют три возможности: вставить отдельный символ, вставить определенное количество одного и того же символа и вставить диапазон символов, заданный двумя итераторами (см. раздел 13.2.14).

Аналогично функциям `append()` и `insert()`, несколько вариантов функций `erase()` и `pop_back()` (начиная со стандарта C++11) удаляют символы, а несколько вариантов функции `replace()` заменяют символы. Например:

```
std::string s = "i18n";                 // s: i18n
s.replace(1,2,"nternationalizatio");    // s: internationalization
s.erase(13);                            // s: international
s.erase(7,5);                           // s: internal
s.pop_back();                           // s: internal (начиная со стандарта
                                        //    C++11)
s.replace(0,2,"ex");                    // s: external
```

С помощью функции `resize()` можно изменить количество символов. Если новый размер, передаваемый как аргумент, меньше текущего количества символов, то символы удаляются с конца строки. Если новый размер больше текущего количества символов, то символы добавляются в конец строки. Если размер строки увеличивается, функции можно передать добавляемый символ. Если этого не сделать, то будет использоваться конструктор по умолчанию для данного символьного типа (для типа `char` используется символ `'\0'`).

13.2.9. Конкатенация подстрок и строк

Из строки можно извлечь подстроку с помощью функции-члена `substr()`. Например:

```
std::string s("interchangeability");
s.substr()                // возвращает копию s
s.substr(11)              // возвращает string("ability")
s.substr(5,6)             // возвращает string("change")
s.substr(s.find('c'))     // возвращает string("changeability")
```

Для конкатенации двух строки или С-строк, а также строк и С-строк с одной стороны и одиночного символа с другой можно использовать оператор +. Например, операторы

```
std::string s1("enter");
std::string s2("nation");
std::string i18n;

i18n = 'i' + s1.substr(1) + s2 + "aliz" + s2.substr(1);
std::cout << "i18n means: " + i18n << std::endl;
```

выводят следующий результат:

```
i18n means: internationalization
```

После принятия стандарта C++11 оператор + имеет перегруженный вариант для строк, использующий rvalue-ссылки для поддержки семантики перемещения. Таким образом, если строковый аргумент, передаваемый оператору +, впоследствии будет не нужен, то для его передачи следует использовать функцию `move()`. Например:

```
string foo()
{
   std::string s1("international");
   std::string s2("ization");
   std::string s = std::move(s1) + std::move(s2); // OK
   // s1 и s2 имеют корректное состояние с неопределенным значением
   return s;
}
```

13.2.10. Операторы ввода-вывода

Для строк определены обычные операторы ввода-вывода.

- **Оператор >>** читает строку из потока ввода.
- **Оператор <<** записывает строку в поток вывода.

Эти операторы действуют так же, как и для обычных С-строк. В частности, оператор >> действует следующим образом.

1. Если установлен флаг `skipws`, то ведущие пробельные символы игнорируются (см. раздел 15.7.7).
2. Символы считываются, пока не произойдет одно из следующих событий:

- следующий символ является пробельным;
- поток перешел в некорректное состояние (например, достигнут конец файла);
- текущее значение `width()` для потока (см. раздел 15.7.3) больше `0`, и считано `width()` символов.
- считано `max_size()` символов.

3. Значение `width()` для потока устанавливается равным `0`.

Таким образом, оператор ввода считывает следующее слово, игнорируя начальные пробельные символы. Пробельным считается любой символ, для которого выражение `isspace(c,`*strm*`.getloc())` имеет значение `true` (функция `isspace()` описана в разделе 16.4.4).

Оператор вывода помимо этого учитывает значение `width()` для потока. Если `width()` больше `0`, то оператор << записывает не менее `width()` символов.

Отметим, что после принятия стандарта C++11 операторы << и >> могут обрабатывать rvalue-ссылки на потоки. Это, например, позволяет использовать временные строковые потоки (подробнее об этом — в разделе 15.10.2).

Функция `getline()`

Строковые классы содержат также специальную вспомогательную функцию `std::getline()` для последовательного чтения строк: эта функция считывает все символы, включая ведущие пробельные, пока не будет достигнут конец строки или файла. Разделитель строк извлекается из потока, но не добавляется в строку. По умолчанию разделителем строк считается символ новой строки, но пользователь может передать свой разделитель строк в виде необязательного аргумента[10]. Таким образом, можно последовательно считывать лексемы, разделенные произвольным символом.

```
std::string s;

while (getline(std::cin,s)) {     // для каждой строки из потока cin
   ...
}

while (getline(std::cin,s,':')) { // для каждой лексемы, отделенной символом ':'
   ...
}
```

Отметим, что при последовательном считывании лексем символ перехода на новую строку не считается специальным символом. В этом случае лексемы могут содержать символ перехода на новую строку.

Отметим также, что после принятия стандарта C++11 функция `getline()` перегружена как для lvalue-, так и для rvalue-ссылок на поток. Это позволяет использовать временные строковые потоки.

```
void process (const std::string& filecontents)
{
```

[10] Перед функцией `getline()` не требуется указывать квалификатор `std::`, потому что при вызове функции поиск, зависящий от аргументов (*argument dependent lookup* — ADL, известный также как *поиск Кёнига*), всегда учитывает пространство имен, в котором определен класс аргумента.

```
    // обрабатываем первую строку из переданного объекта класса string:
    std::string firstLine;
    std::getline(std::stringstream(filecontents), // OK, начиная
                                                  // со стандарта C++11
                 firstLine);
    ...
}
```

Подробное описание строковых потоков приведено в разделе 15.10.

13.2.11. Поиск

В стандартной библиотеке C++ есть много средств для поиска подстрок в строке.

- С помощью **функций-членов** можно выполнять поиск:
  - отдельного символа, последовательности символов (подстроку) или определенного множества символов;
  - в прямом и обратном направлении;
  - начиная с любой позиции в начале или в середине строки.
- Используя библиотеку регулярных выражений (**regex**) (см. главу 14), можно искать более сложные шаблоны символьных последовательностей. Пример приведен в разделе 13.2.14.
- Используя **алгоритмы STL**, можно искать отдельные символы или заданные последовательности символов (см. раздел 11.2.2). Отметим, что эти алгоритмы позволяют применять пользовательские критерии сравнения (см. пример в разделе 13.2.14).

Функции-члены поиска

Все функции поиска содержат в своем названии слово *find*. Они пытаются найти позицию символа для заданного значения *value*, которое передается как аргумент. Выполнение поиска зависит от точного имени функции. В табл. 13.5 перечислены все функции поиска для строк.

Таблица 13.5. Функции поиска для строк

Строка	Действие
`find()`	Выполняет поиск первого вхождения *value*
`rfind()`	Выполняет поиск последнего вхождения *value* (поиск в обратном направлении)
`find_first_of()`	Выполняет поиск первого символа, который является частью *value*
`find_last_of()`	Выполняет поиск последнего символа, который является частью *value*
`find_first_not_of()`	Выполняет поиск первого символа, не являющегося частью *value*
`find_last_not_of()`	Выполняет поиск последнего символа, не являющегося частью *value*

Все функции поиска возвращают индекс первого символа в символьной последовательности, который соответствует критерию поиска. Если поиск завершается неудачей, функции возвращают значение `npos`. Функции поиска используют следующую схему аргументов.

- Первый аргумент всегда является искомой подстрокой.
- Второе необязательное значение задает индекс, с которого начинается поиск в строке.
- Необязательный третий аргумент — это количество символов в искомой подстроке.

К сожалению, эта схема аргументов отличается от схемы остальных функций для работы со строками. В других функциях для работы со строками начальный индекс задается первым аргументом, а подстрока и ее длина — следующими аргументами. В частности, каждая функция поиска перегружена для следующих наборов аргументов.

- `const string& value`

  Выполняет поиск символов строки *value*.

- `const string& value, size_type idx`

  Выполняет поиск символов строки *value*, начиная с индекса *idx* в строке `*this`.

- `const char* value`

  Выполняет поиск символов С-строки *value*.

- `const char* value, size_type idx`

  Выполняет поиск символов С-строки *value*, начиная с индекса *idx* в строке `*this`.

- `const char* value, size_type idx, size_type value_len`

  Выполняет поиск *value_len* символов из символьного массива *value*, начиная с индекса *idx* в строке `*this`. Таким образом, нулевой символ (`'\0'`) *не имеет* специального смысла в строке *value*.

- `const char value`

  Выполняет поиск символа *value*.

- `const char value, size_type idx`

  Выполняет поиск символов *value*, начиная с индекса *idx* в строке `*this`.

Рассмотрим пример:

```
std::string s("Hi Bill, I'm ill, so please pay the bill");

s.find("il")           // возвращает 4 (первая подстрока "il")
s.find("il",10)        // возвращает 13 (первая подстрока "il" начиная с s[10])
s.rfind("il")             // возвращает 37 (последняя подстрока "il")
s.find_first_of("il")     // возвращает 1 (первый символ 'i' или 'l')
s.find_last_of("il")      // возвращает 39 (последний символ 'i' или 'l')
s.find_first_not_of("il") // возвращает 0 (первый символ,
                          // не равный ни 'i', ни 'l')
s.find_last_not_of("il")  // возвращает  36 (последний символ,
                          // не равный ни 'i', ни 'l')
s.find("hi")              // возвращает npos
```

Отметим, что схема именования алгоритмов поиска в библиотеке STL отличается от схемы именования функций поиска в строках (см. раздел 11.2.2).

13.2.12. Значение npos

Если поиск завершается неудачей, функция возвращает значение *string*::npos. Рассмотрим следующий пример:

```
std::string s;
std::string::size_type idx; // осторожно: не используйте другой тип!
...
idx = s.find("substring");
if (idx == std::string::npos) {
    ...
}
```

Условие операции if выполняется тогда и только тогда, когда "substring" не является частью строки s.

Используя строковое значение npos и его тип, следует быть очень осторожным. Если требуется проверить возвращаемое значение, для типа возвращаемого значения всегда следует использовать тип string::size_type, *а не* int или unsigned, в противном случае сравнение возвращаемого значения со значением string::npos может оказаться некорректным. Это объясняется тем, что разработчики библиотеки определили значение npos равным -1.

```
namespace std {
   template <typename charT,
             typename traits = char_traits<charT>,
             typename Allocator = allocator<charT> >
   class basic_string {
      public:
         typedef typename Allocator::size_type size_type;
         ...
         static const size_type npos = -1;
         ...
   };
}
```

К сожалению, тип size_type, определенный механизмом распределения памяти для строк, должен быть целочисленным типом без знака. Механизм распределения памяти, предусмотренный по умолчанию, allocator, использует в качестве типа size_type тип size_t. Поскольку -1 преобразуется в целочисленный тип без знака, значение npos является максимальным значением без знака для этого типа. Однако точное значение зависит от точного определения типа size_type. К сожалению, эти максимальные значения могут оказаться разными. Фактически (unsigned long)-1 *отличается* от (unsigned short)-1, если размеры типов разные. Таким образом, сравнение

```
idx == std::string::npos
```

может вернуть false, если idx равно -1 и к тому же idx и string::npos имеют разные типы:

```
std::string s;
...
int idx = s.find("not found");  // допустим, что возвращается npos
if (idx == std::string::npos) { // ОШИБКАR: некорректное Сравнение
...
}
```

Избежать этой ошибки можно, например, проверяя успешность поиска непосредственно:

```
if (s.find("hi") == std::string::npos) {
   ...
}
```

Однако часто необходим индекс позиции, в которой найдено совпадение символов. Таким образом, другое простое решение заключается в определении собственного значения npos со знаком.

```
const int NPOS = -1;
```

Теперь сравнение выглядит несколько иначе и даже более удобным:

```
if (idx == NPOS) { // работает практически всегда
   ...
}
```

К сожалению, это решение не идеально, потому что сравнение окажется некорректным, если либо переменная idx имеет тип unsigned short, либо индекс превышает максимальное значение типа int. Из-за этого разработчики стандарта отказались от него. Однако, поскольку обе эти ситуации встречаются довольно редко, такое решение является работоспособным в большинстве ситуаций. Для того чтобы написать переносимый код, всегда следует использовать тип *string*::size_type для любого индекса строкового типа. Для идеального решения задачи следовало бы перегрузить функции для точного типа string::size_type. Возможно, в будущем разработчики стандарта примут более удачное решение (впрочем, в стандарте C++11 ничего не изменилось).

13.2.13. Числовые преобразования

В соответствии со стандартом C++11 стандартная библиотека C++ содержит вспомогательные функции для преобразования строк в числовые значения, и наоборот (см. табл. 13.6). Тем не менее эти преобразования предусмотрены только для типов string и wstring, но не u16string и u32string.

Таблица 13.6. Преобразования чисел в строки

Строковая функция	Действие
stoi(*str*, *idxRet*=nullptr, *base*=10)	Преобразует *str* в тип int
stol(*str*, *idxRet*=nullptr, *base*=10)	Преобразует *str* в тип long
stoul(*str*, *idxRet*=nullptr, *base*=10)	Преобразует *str* в тип unsigned long
stoll(*str*, *idxRet*=nullptr, *base*=10)	Преобразует *str* в тип long long
stoull(*str*, *idxRet*=nullptr, *base*=10)	Преобразует *str* в тип unsigned long long
stof(*str*, *idxRet*=nullptr)	Преобразует *str* в тип float
stod(*str*, *idxRet*=nullptr)	Преобразует *str* в тип double
stold(*str*, *idxRet*=nullptr)	Преобразует *str* в тип long double
to_string(*val*)	Преобразует *val* в тип string
to_wstring(*val*)	Преобразует *val* в тип wstring

Все функции, преобразующие строки в числовые значения, подчиняются следующим правилам:

- игнорируют ведущие пробельные символы;
- позволяют возвращать индекс первого символа, стоящего после последнего обработанного символа;
- могут генерировать исключения `std::invalid_argument`, если преобразование невозможно, и `std::out_of_range`, если преобразованное значение выходит за пределы допустимого диапазона для возвращаемого значения;
- для целочисленных значений можно передавать необязательный аргумент — основание системы счисления.

Для всех функций, преобразующих числовое значение в `string` или `wstring`, значение *val* может иметь один из следующих типов: `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, `double` или `long double`.

Например, рассмотрим следующую программу:

```
// string/stringnumconv1.cpp

#include <string>
#include <iostream>
#include <limits>
#include <exception>

int main()
{
   try {
      // преобразование в числовой тип
      std::cout << std::stoi (" 77") << std::endl;
      std::cout << std::stod (" 77.7") << std::endl;
      std::cout << std::stoi ("-0x77") << std::endl;

      // используем индексы необработанных символов
      std::size_t idx;
      std::cout << std::stoi (" 42 is the truth", &idx) << std::endl;
      std::cout << " idx of first unprocessed char: " << idx << std::endl;

      // используем основания 16 и 8
      std::cout << std::stoi (" 42", nullptr, 16) << std::endl;
      std::cout << std::stol ("789", &idx, 8) << std::endl;
      std::cout << " idx of first unprocessed char: " << idx << std::endl;

      // преобразуем числовое значение в строку
      long long ll = std::numeric_limits<long long>::max();
      std::string s = std::to_string(ll);    // преобразуем максимальный
                                             // long long в строку
      std::cout << s << std::endl;

      // пытаемся выполнить обратное преобразование
      std::cout << std::stoi(s) << std::endl; // генерируется out_of_range
   }
   catch (const std::exception& e) {
```

```
        std::cout << e.what() << std::endl;
    }
}
```

Программа выводит следующий результат:

```
77
77.7
0
42
idx of first unprocessed char: 4
66
7
idx of first unprocessed char: 1
9223372036854775807
stoi argument out of range
```

Отметим, что выражение `std::stoi("-0x77")` равно `0`, потому что оно распознает только `-0`, интерпретируя `x` как конец обнаруженного числового значения. Выражение `std::stol("789",&idx,8)` распознает только первый символ строки, потому что `8` не является допустимым символом для восьмеричных чисел.

13.2.14. Поддержка итераторов для строк

Строка — это упорядоченная последовательность символов. По этой причине стандартная библиотека C++ содержит интерфейс для строк, позволяющий использовать их как контейнеры STL[11].

В частности, можно вызывать обычные функции-члены, чтобы получить итераторы, выполняющие обход символов в строке. Если итераторы вам не знакомы, рассматривайте их как нечто, что может ссылаться на отдельный символ в середине строки, аналогично обычным указателям для С-строк. Используя эти объекты, можно пройти по всем символам строки, вызывая несколько алгоритмов, предусмотренных в стандартной библиотеке C++ либо определенных пользователем. Например, можно упорядочить символы строки, изменить их порядок или найти символ, имеющий максимальное значение.

Строковые итераторы являются итераторами произвольного доступа. Это значит, что они обеспечивают произвольный доступ к символам и допускают использование всех алгоритмов (категории итераторов обсуждались в разделах 6.3.2 и 9.2). Как обычно, типы строковых итераторов (`iterator`, `const_iterator` и т.д.) определяются самим строковым классом. Точный тип зависит от реализации, но обычно строковые итераторы определяются как обычные указатели. Огромная разница между итераторами, реализованными как указатели, и итераторами, реализованными в виде классов, обсуждались в разделе 9.2.6.

После перераспределения памяти или изменения значений, на которые они ссылаются, итераторы становятся некорректными. Подробности изложены в разделе 13.2.6.

Функции для работы со строковыми итераторами

В табл. 13.7 приведены все функции-члены строковых классов для работы с итераторами. Как обычно, диапазон, заданный итераторами `beg` и `end`, — это полуоткрытый диапазон, который включает позицию `beg`, но не содержит позиции `end` и записывается как `[beg,end)` (см. раздел 6.3).

[11] Библиотека STL описана в главе 6.

Таблица 13.7. Строковые операции над итераторами

Выражение	Действие
`s.begin(), s.cbegin()`	Возвращает итератор произвольного доступа, установленный на первый символ
`s.end(), s.cend()`	Возвращает итератор произвольного доступа, установленный на последний символ
`s.rbegin(), s.crbegin()`	Возвращает обратный итератор, установленный на первый символ при обратном обходе (т.е. последний символ при прямом обходе)
`s.rend(), crend()`	Возвращает обратный итератор, установленный на позицию, следующую за последним символом при обратном обходе (т.е. на позицию, предшествующую первому символу при прямом обходе)
`string s(`*beg*`,`*end*`)`	Создает строку, инициализированную всеми символами диапазона [*beg*,*end*)
`s.append(`*beg*`,`*end*`)`	Добавляет все символы диапазона [*beg*,*end*)
`s.assign(`*beg*`,`*end*`)`	Присваивает все символы диапазона [*beg*,*end*)
`s.insert(`*pos*`,`*c*`)`	Вставляет символ *c* в позицию итератора *pos* и возвращает позицию итератора, установленного на новый символ
`s.insert(`*pos*`,`*num*`,`*c*`)`	Вставляет *num* вхождений символа *c* в позицию итератора *pos* и возвращает позицию итератора, установленного на первый новый символ
`s.insert(`*pos*`,`*beg*`,`*end*`)`	Вставляет все символы диапазона [*beg*,*end*) в позицию итератора *pos*
`s.insert(`*pos*`,`*initlist*`)`	Вставляет все символы списка инициализации *initlist* в позицию итератора *pos* (начиная со стандарта C++11)
`s.erase(`*pos*`)`	Удаляет символ, на который ссылается итератор *pos*, и возвращает позицию следующего символа
`s.erase(`*beg*`,`*end*`)`	Удаляет все символы диапазона [*beg*,*end*) и возвращает позицию следующего символа
`s.replace(`*beg*`,`*end*`,`*str*`)`	Заменяет все символы диапазона [*beg*,*end*) символами строки *str*
`s.replace(`*beg*`,`*end*`,`*cstr*`)`	Заменяет все символы диапазона [*beg*,*end*) символами С-строки *cstr*
`s.replace(`*beg*`,`*end*`,`*cstr*`,`*len*`)`	Заменяет все символы диапазона [*beg*,*end*) *len* символами массива символов *cstr*
`s.replace(`*beg*`,`*end*`,`*num*`,`*c*`)`	Заменяет все символы диапазона [*beg*,*end*) *num* вхождений символа *c*
`s.replace(`*beg*`,`*end*`,`*newBeg*`,`*newEnd*`)`	Заменяет все символы диапазона [*beg*,*end*) всеми символами диапазона [*newBeg*,*newEnd*)
`s.replace(`*beg*`,`*end*`,`*initlist*`)`	Заменяет все символы диапазона [*beg*,*end*) значениями списка инициализации *initlist* (начиная со стандарта C++11)

Для поддержки итераторов вставки в конец предназначена функция `push_back()`. Описание итераторов вставки в конец и пример их использования для работы со строками см. в разделе 9.4.2.

Пример использования строковых итераторов

С помощью строковых итераторов можно перевести все символы в верхний или нижний регистр с помощью всего одного оператора.

```
// string/stringiter1.cpp

#include <string>
#include <iostream>
#include <algorithm>
#include <cctype>
#include <regex>
using namespace std;

int main()
{
   // создаем строку
   string s("The zip code of Braunschweig in Germany is 38100");
   cout << "original: " << s << endl;

   // переводим все символы в нижний регистр
   transform (s.cbegin(), s.cend(), // источник
              s.begin(),            // получатель
              [] (char c) {         // операция
                  return tolower(c);
              });
   cout << "lowered: " << s << endl;

   // переводим все символы в верхний регистр
   transform (s.cbegin(), s.cend(), // источник
              s.begin(),            // получатель
              [] (char c) { // operation
                  return toupper(c);
              });
   cout << "uppered: " << s << endl;

   // выполняем поиск строки "Germany" без учета регистра
   string g("Germany");
   string::const_iterator pos;
   pos = search (s.cbegin(),s.cend(),    // строка, в которой идет поиск
                 g.cbegin(),g.cend(),    // искомая подстрока
                 [] (char c1, char c2) { // критерий сравнения
                 return toupper(c1) == toupper(c2);
   });
   if (pos != s.cend()) {
       cout << "substring \"" << g << "\" found at index "
            << pos - s.cbegin() << endl;
   }
}
```

Здесь дважды использованы итераторы, возвращаемые функциями `cbegin()`, `cend()` и `begin()`. Они передаются алгоритму `transform()`, который преобразовывает все элементы входного диапазона в диапазон-получатель, используя операцию, заданную как четвертый аргумент (см. разделы 6.8.1 и 11.6.3).

Преобразование задается лямбда-функцией (см. раздел 6.9), которая преобразовывает элементы строки (символы) в нижний или верхний регистр. Отметим, что функции `tolower()` и `toupper()` представляют собой устаревшие функции языка C, использующие универсальный локальный контекст. Если в программе используется другой локальный контекст или несколько локальных контекстов, то следует использовать новую форму функций `tolower()` и `toupper()`. Детали изложены в разделе 16.4.4.

В заключение мы используем алгоритм поиска подстроки по собственному критерию. Этот критерий задан лямбда-функцией, сравнивающей символы без учета регистра.

В качестве альтернативы можно было бы использовать библиотеку регулярных выражений.

```
// поиск строки "Germany" без учета регистра
std::regex pat("Germany",                  // искомое выражение
               regex_constants::icase);    // поиск без учета регистра
smatch m;
if (regex_search (s,m,pat)) {              // ищем регулярный шаблон в s
    cout << "substring \"Germany\" found at index "
         << m.position() << endl;
}
```

Подробности изложены в разделе 14.6.

Таким образом, программа выводит следующий результат:

```
original: The zip code of Braunschweig in Germany is 38100
lowered: the zip code of braunschweig in germany is 38100
uppered: THE ZIP CODE OF BRAUNSCHWEIG IN GERMANY IS 38100
substring "Germany" found at index 32
```

В последнем операторе вывода можно вычислить разность между двумя строковыми итераторами, чтобы найти индекс позиции символа:

```
pos - s.cbegin()
```

Здесь использован оператор -, который можно использовать, потому что итераторы относятся к категории итераторов произвольного доступа. Аналогично для преобразования индекса в позицию итератора можно просто добавить значение этого индекса.

При работе с множествами или отображениями строк может понадобиться специальный критерий сортировки строк без учета регистра. Соответствующий пример приведен в разделе 7.8.6.

Следующая программа демонстрирует еще один пример использования функций для работы со строковыми итераторами:

```
// string/stringiter2.cpp

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

```
int main()
{
   // создаем константную строку
   const string hello("Hello, how are you?");

   // инициализируем строку s всеми символами строки hello
   string s(hello.cbegin(),hello.cend());

   // диапазонный цикл обхода всех символов
   for (char c : s) {
       cout << c;
   }
   cout << endl;

   // изменяем на обратный порядок всех символов внутри строки
   reverse (s.begin(), s.end());
   cout << "reverse:       " << s << endl;

   // сортируем все символы внутри строки
   sort (s.begin(), s.end());
   cout << "ordered:       " << s << endl;

   // удаляем соседние дубликаты
   // - unique() изменяет порядок и возвращает новый конец
   // - erase()  уменьшает размер
   s.erase (unique(s.begin(),
                   s.end()),
                   s.end());
   cout << "no duplicates: " << s << endl;
}
```

Эта программа выводит следующий результат:

```
Hello, how are you?
reverse:       ?uoy era woh ,olleH
ordered:          ,?Haeehlloooruwy
no duplicates:  ,?Haehloruwy
```

Следующий пример демонстрирует использование итераторов вставки в конец для считывания стандартного потока в строку:

```
// string/string3.cpp

#include <string>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <locale>
using namespace std;

int main()
{
   string input;
```

```
    // не игнорируем начальные пробельные символы
    cin.unsetf (ios::skipws);

    // читаем все символы, сжимая пропуски
    const locale& loc(cin.getloc());          // локальный контекст
    unique_copy(istream_iterator<char>(cin),  // начало источника
                istream_iterator<char>(),     // конец источника
                back_inserter(input),         // получатель
                [=] (char c1, char c2) {      // критерий для смежных
                                              // дубликатов
    return isspace(c1,loc) && isspace(c2,loc);
    });

    // обработка ввода
    // - записываем в стандартный поток вывода
    cout << input;
}
```

Используя алгоритм `unique_copy()` (см. раздел 11.7.2), мы считываем все символы из входного потока `cin` и вставляем их в строку `input`.

Передаваемая лямбда-операция проверяет, являются ли два символа пробельными. Этот критерий используется алгоритмом `unique_copy()` для выявления смежных дубликатов и удаления второго дубликата. Таким образом, читая ввод, алгоритм сжимает многократные пробельные символы (функция `isspace()` обсуждается в разделе 16.4.4).

Критерий учитывает локальный контекст. Для этого переменная `loc` инициализируется локальным контекстом `cin` и передается по значению лямбда-функции (функция `getloc()` рассматривается в разделе 15.8).

Аналогичный пример можно найти в разделе 11.7.2, посвященном алгоритму `unique_copy()`.

13.2.15. Интернационализация

Как указано в разделе 13.2.1, параметрами шаблонного строкового класса `basic_string<>` являются символьный тип, свойства символьного типа и модель памяти. Тип `string` — это специализация символов типа `char`, в то время как типы `wstring`, `u16string` и `u32string` представляют собой специализации для символов типов `wchar_t`, `char16_t` и `char32_t` соответственно.

После принятия стандарта C++11 пользователь может самостоятельно задавать наборы символов, используемых как строковые литералы (см. раздел 3.1.6).

Для уточнения аспектов, зависящих от представления символьного типа, используются свойства символов. Дополнительный класс необходим, потому что интерфейс встроенных типов, таких как `char` и `wchar_t`, изменить невозможно, причем один и тот же символьный тип может иметь разные свойства. Подробное описание классов свойств приведено в разделе 16.1.4.

В следующей программе определен специальный класс свойств для строк, дающий возможность работать с ними без учета регистра символов:

```
// string/icstring.hpp

#ifndef ICSTRING_HPP
```

```
#define ICSTRING_HPP

#include <string>
#include <iostream>
#include <cctype>

// заменяем функции стандартного класса char_traits<char>,
// чтобы операции над строками не зависели от регистра
struct ignorecase_traits : public std::char_traits<char> {
   // проверяем равенство c1 и c2
   static bool eq(const char& c1, const char& c2) {
      return std::toupper(c1)==std::toupper(c2);
   }
   // проверяем условие "c1 меньше c2"
   static bool lt(const char& c1, const char& c2) {
      return std::toupper(c1)<std::toupper(c2);
   }
   // сравниваем до n символов строк s1 и s2
   static int compare(const char* s1, const char* s2,
                      std::size_t n) {
      for (std::size_t i=0; i<n; ++i) {
          if (!eq(s1[i],s2[i])) {
              return lt(s1[i],s2[i])?-1:1;
          }
      }
      return 0;
   }
   // ищем символ c в строке s
   static const char* find(const char* s, std::size_t n,
                           const char& c) {
      for (std::size_t i=0; i<n; ++i) {
          if (eq(s[i],c)) {
              return &(s[i]);
          }
      }
      return 0;
   }
};

// определяем специальный тип для таких строк
typedef std::basic_string<char,ignorecase_traits> icstring;

// определяем операторы вывода
// потому что тип свойств отличается от такового для std::ostream
inline
std::ostream& operator << (std::ostream& strm, const icstring& s)
{
   // просто преобразовываем icstring в обычную строку
   return strm << std::string(s.data(),s.length());
}
#endif    // ICSTRING_HPP
```

Определение оператора вывода необходимо, потому что стандарт определяет операторы ввода-вывода только для потоков, у которых символьный тип и тип свойств совпадают. Однако в данном случае типы свойств отличаются друг от друга, поэтому мы должны определить свой оператор вывода. Это относится и к операторам ввода.

Следующая программа демонстрирует использование определенных выше специальных видов строк:

```
// string/icstring1.cpp

#include "icstring.hpp"
int main()
{
   using std::cout;
   using std::endl;

   icstring s1("hallo");
   icstring s2("otto");
   icstring s3("hALLo");

   cout << std::boolalpha;
   cout << s1 << " == " << s2 << " : " << (s1==s2) << endl;
   cout << s1 << " == " << s3 << " : " << (s1==s3) << endl;

   icstring::size_type idx = s1.find("All");
   if (idx != icstring::npos) {
       cout << "index of \"All\" in \"" << s1 << "\": "
            << idx << endl;
   }
   else {
       cout << "\"All\" not found in \"" << s1 << endl;
   }
}
```

Программа выводит следующий результат:

```
hallo == otto : false
hallo == hALLo : true
index of "All" in "hallo": 1
```

Более подробно вопросы, связанные с интернационализацией, описаны в главе 16.

13.2.16. Производительность

Как обычно, стандарт *не* определяет, *как* реализовать строковый класс, регламентируя только его интерфейс. В зависимости от основных принципов и приоритетов реализации могут сильно отличаться по быстродействию и использованию памяти.

Отметим, что после принятия стандарта C++11 реализации с подсчетом ссылок больше не допускаются. Это объясняется тем, что реализации, позволяющие строкам использовать общие внутренние буферы, не работают в многопоточных средах.

13.2.17. Строки и векторы

Строки и векторы функционируют аналогично. Это не удивительно, потому что оба класса представляют собой контейнеры, которые обычно реализуются с помощью динамических массивов. Таким образом, строку следует интерпретировать как разновидность вектора, элементами которого являются символы. Фактически строку можно использовать как контейнер STL (см. раздел 13.2.14). Однако такая интерпретация опасна, потому что между строками и векторами существуют фундаментальные отличия. Среди них можно выделить два основных различия.

1. Основная цель векторов — манипулирование элементами контейнера, а не контейнером в целом. Таким образом, реализации векторов оптимизируются для работы с элементами внутри контейнера.
2. Основная цель строк — манипулирование контейнером (строкой) в целом.

Таким образом, строки оптимизируются так, чтобы сократить затраты на выполнение операторов присваивания и передачи всего контейнера. Обычно это приводит к созданию совершенно разных реализаций. Тем не менее векторы можно использовать как обычные С-строки (см. раздел 7.3.3).

13.3. Подробное описание класса `string`

В этом разделе слово ***`string`*** означает соответствующий строковый класс: `string`, `wstring`, `u16string`, `u32string` или любую другую специализацию класса `basic_string<>`. Тип ***`char`*** означает соответствующий символьный тип, т.е. `char` для `string`, `wchar_t` для `wstring`, `char16_t` для `u16string` или `char32_t` для `u32string`. Определения других типов и значений, выделенных курсивом, зависят от индивидуальных определений символьного типа или класса свойств. Подробное описание классов свойств приведено в разделе 16.1.4.

13.3.1. Определения типов и статические значения

***`string`*`::traits_type`**

- Тип символьных свойств.
- Второй шаблонный параметр класса `basic_string`.
- Для типа `string` эквивалентно классу `char_traits<char>`.

***`string`*`::value_type`**

- Символьный тип.
- Эквивалент `traits_type::char_type`.
- Для типа `string` эквивалентно `char`.

***`string`*`::size_type`**

- Целочисленный тип без знака для значений размера и индексов.
- Эквивалент `allocator_type::size_type`.
- Для типа `string` эквивалентно `size_t`.

***string*::difference_type**

- Целочисленный тип со знаком для значений разности.
- Эквивалент `allocator_type::difference_type`.
- Для типа `string` эквивалентно `ptrdiff_t`.

***string*::reference**

- Тип ссылок на символы.
- Эквивалент `allocator_type::reference`.
- Для типа `string` эквивалентно `char&`.

***string*::const_reference**

- Тип константных ссылок на символы.
- Эквивалент `allocator_type::const_reference`.
- Для типа `string` эквивалентно `const char&`.

***string*::pointer**

- Тип указателей на символы.
- Эквивалент `allocator_type::pointer`.
- Для типа `string` эквивалентно `char*`.

***string*::const_pointer**

- Тип константных указателей на символы.
- Эквивалент `allocator_type::const_pointer`.
- Для типа `string` эквивалентно `const char*`.

***string*::iterator**

- Тип итераторов.
- Точный тип зависит от реализации.
- Для типа `string` обычно `char*`.

***string*::const_iterator**

- Тип константных итераторов.
- Точный тип зависит от реализации.
- Для типа `string` обычно `const char*`.

***string*::reverse_iterator**

- Тип обратных итераторов.
- Эквивалент `reverse_iterator<iterator>`.

***string*::const_reverse_iterator**

- Тип константных обратных итераторов.
- Эквивалент `reverse_iterator<const_iterator>`.

`static const` *size_type* **`string::npos`**

- Специальное значение, означающее "*не найден*" или "*все оставшиеся символы*".
- Целочисленное значение без знака, инициализированное значением `-1`.
- Значение `npos` следует использовать с осторожностью (см. раздел 13.2.12).

13.3.2. Операции создания, копирования и уничтожения

`string::string` `()`

- Конструктор по умолчанию.
- Создает пустую строку.

`string::string` `(const` *string&* *str*`)`

- Копирующий конструктор.
- Создает новую строку как копию аргумента *str*.

`string::string` `(`*string&&* *str*`)`

- Перемещающий конструктор.
- Создает новую строку, инициализированную элементами существующей строки `str`.
- После операции содержимое строки `str` становится неопределенным.
- Доступно начиная со стандарта C++11.

`string::string` `(const` *string&* *str*`, size_type` *str_idx*`)`
`string::string` `(const` *string&* *str*`, size_type` *str_idx*`,`
`size_type` *str_num*`)`

- Создают новую строку, инициализированную не более *str_num* первых символов строки *str*, начиная с индекса *str_idx*.
- Если аргумент *str_num* не указан, то используются все символы от *str_idx* до конца строки *str*.
- Генерирует исключение `out_of_range`, если *str_idx* `>` *str*`.size()`.

`string::string` `(const` *char** *cstr*`)`

- Создает строку, инициализированную С-строкой *cstr*.
- Строка инициализируется всеми символами С-строки *cstr*, исключая символ `'\0'`.
- Передача нулевого указателя (`nullptr` или `NULL`) приводит к непредсказуемым последствиям.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

`string::string` `(const` *char** *chars*`, size_type` *chars_len*`)`

- Создает строку, инициализированную *chars_len* символами массива *chars*.
- Массив *chars* должен содержать как минимум *chars_len* символов. Символы могут быть любыми. Таким образом, символ `'\0'` не имеет специального значения.

- Генерирует исключение `length_error`, если *chars_len* равно *string*::`npos`.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

***string*::string** (`size_type` *num*, *char c*)

- Создает строку, инициализированную *num* экземплярами символа *c*.
- Генерирует исключение `length_error`, если *num* равно *string*::`npos`.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

***string*::string** (`InputIterator` *beg*, `InputIterator` *end*)

- Создает строку, инициализированную всеми символами диапазона [*beg*,*end*).
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

***string*::string** (*initializer-list*)

- Создает новую строку, инициализированную символами списка инициализации *initializer-list*.
- Доступно начиная со стандарта C++11.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

***string*::~string** ()

- Деструктор.
- Уничтожает все символы и освобождает память.

Большинство конструкторов позволяют передавать в качестве дополнительного аргумента механизм распределения памяти (см. раздел 13.3.13).

13.3.3. Операции над размерами и емкостью

Операции над размерами

`bool` ***string*::empty** () `const`

- Проверяет, пустая ли строка (не содержит ни одного символа).
- Эквивалент *string*::`size()==0`, но может работать быстрее.

`size_type` ***string*::size** () `const`
`size_type` ***string*::length** () `const`

- Обе функции возвращают текущее количество символов.
- Обе функции эквивалентны.
- Для проверки, пустая ли строка, следует использовать функцию `empty()`, потому что она может работать быстрее.

`size_type` ***`string`*`::max_size`** `() const`

- Возвращает максимальное количество символов, которое может содержать строка.
- Если длина строки, являющейся результатом операции, превышает `max_size()`, класс генерирует исключение `length_error`.

Операции над емкостью

`size_type` ***`string`*`::capacity`** `() const`

- Возвращает количество символов, которое строка может содержать без перераспределения памяти.

`void` ***`string`*`::reserve`** `()`
`void` ***`string`*`::reserve`** `(size_type` *num*`)`

- Первая форма является необязательным запросом на сжатие до реального количества символов.
- Вторая форма резервирует внутреннюю память размером не менее *num* символов.
- Если *num* меньше текущей емкости, то вызов интерпретируется как необязательный запрос на уменьшение емкости.
- Если *num* меньше текущего количества символов, то вызов интерпретируется как необязательный запрос на уменьшение емкости до количества символов (эквивалент первой формы).
- Емкость никогда не становится меньше текущего количества символов.
- Операция может делать некорректными ссылки, указатели и итераторы, установленные на символы. Однако стандарт гарантирует, что перераспределение не происходит во время вставок после выполнения функции `reserve()`, пока очередная вставка не приведет к тому, что размер станет больше `num`. Таким образом, функция `reserve()` может увеличивать быстродействие и помогает сохранять ссылки, указатели и итераторы корректными (см. раздел 13.2.5).

`void` ***`string`*`::shrink_to_fit`** `()`

- Уменьшает внутреннюю память до текущего количества символов.
- Аналог `reserve(0)`.
- Вызов интерпретируется как необязательный запрос на оптимизацию, зависящую от реализации.
- Операция может делать некорректной ссылки, указатели и итераторы, установленные на символы.
- Доступно начиная со стандарта C++11.

13.3.4. Сравнения

`bool` ***`comparison`*** `(const` *`string&`* *`str1`*`, const` *`string&`* *`str2`*`)`
`bool` ***`comparison`*** `(const` *`string&`* *`str`*`, const` *`char*`* *`cstr`*`)`
`bool` ***`comparison`*** `(const` *`char*`* *`cstr`*`, const` *`string&`* *`str`*`)`

- Первая форма возвращает результат сравнения двух строк.
- Вторая и третья формы возвращают результат сравнения строки и С-строки.
- Слово *comparison* можно заменить одним из следующих операторов:

```
operator ==
operator !=
operator <
operator >
operator <=
operator >=
```

- Значения сравниваются лексикографически (см. раздел 13.2.7).

```
int string::compare (const string& str) const
```

- Сравнивает строку `*this` со строкой *str*.
- Возвращает
  - 0, если строки равны;
  - отрицательное значение, если `*this` лексикографически меньше *str*;
  - положительное значение, если `*this` лексикографически больше *str*.
- Для сравнения используется функция `traits::compare()` (подробнее об этом — в разделе 16.1.4).
- См. раздел 13.2.7.

```
int string::compare (size_type idx, size_type len,
                     const string& str) const
```

- Сравнивает не более *len* символов строки `*this`, начиная с индекса *idx*, со строкой *str*.
- Генерирует исключение `out_of_range`, если *idx* > `size()`.
- Сравнение выполняется аналогично функции `compare(str)`.

```
int string::compare (size_type idx, size_type len,
                     const string& str, size_type str_idx,
                     size_type str_len) const
```

- Сравнивает не более *len* символов строки `*this`, начиная с индекса *idx*, не более *str_len* символов строки *str*, начиная с индекса *str_idx*.
- Генерирует исключение `out_of_range`, если *idx* > `size()`.
- Генерирует исключение `out_of_range`, если *str_idx* > *str*`.size()`.
- Сравнение выполняется аналогично функции `compare(str)`.

```
int string::compare (const char* cstr) const
```

- Сравнивает символы строки `*this` с символами С-строки *cstr*.
- Сравнение выполняется аналогично функции `compare(str)`.

```
int string::compare (size_type idx, size_type len,
                     const char* cstr) const
```

- Сравнивает не более *len* символов строки `*this`, начиная с индекса *idx*, со всеми символами С-строки *cstr*.

- Сравнение выполняется аналогично функции `compare(str)`.
- Передача нулевого указателя (`nullptr` или `NULL`) приводит к непредсказуемым последствиям.

```
int string::compare (size_type idx, size_type len,
                     const char* chars, size_type chars_len) const
```

- Сравнивает не более *len* символов строки `*this`, начиная с индекса *idx*, с *chars_len* символами массива *chars*.
- Сравнение выполняется аналогично функции `compare(str)`.
- Массив *chars* должен содержать не менее *chars_len* символов. Символы могут быть любыми. Таким образом, символ `'\0'` не имеет специального значения.
- Генерирует исключение `length_error`, если *chars_len* равно *string*`::npos`.

13.3.5. Доступ к символам

```
char& string::operator[] (size_type idx)
const char& string::operator[] (size_type idx) const
```

- Обе формы возвращают символ с индексом *idx* (первый символ имеет индекс `0`).
- Значения, возвращаемые функцией `length()` или `size()`, являются корректными индексами, причем оператор возвращает значение, сгенерированное конструктором по умолчанию для символьного типа (для `string` этим значением является `'\0'`). До принятия стандарта C++11 значения индексов, для неконстантных строк возвращаемые функцией `length()` или `size()`, были некорректными индексами.
- Передача некорректного индекса приводит к непредсказуемым последствиям.
- При модификации неконстантной строки или перераспределении памяти возвращаемая ссылка может оказаться некорректной (см. раздел 13.2.6).
- Если вызывающая сторона не может гарантировать корректность индекса, то следует использовать функцию `at()`.

```
char& string::at (size_type idx)
const char& string::at (size_type idx) const
```

- Обе формы возвращают символ с индексом *idx* (первый символ имеет индекс `0`).
- Для всех строк индекс, равный `length()`, является некорректным.
- Передача некорректного индекса, т.е. меньшего `0` или большего или равного `length()` или `size()`, генерирует исключение `out_of_range`.
- При модификации неконстантной строки или перераспределении памяти возвращаемая ссылка может оказаться некорректной (см. раздел 13.2.6).
- Если корректность индекса гарантирована, то вызывающая сторона может использовать оператор `[]`, который выполняется быстрее.

```
char& string::front ()
const char& string::front () const
```

- Обе формы возвращают первый символ строки.

- Вызов функции `front()` для пустой строки приводит к возвращению значения, сгенерированного конструктором по умолчанию для символьного типа (для `string` этим значением является `'\0'`).
- При модификации неконстантной строки или перераспределении памяти возвращаемая ссылка может оказаться некорректной (см. раздел 13.2.6).

char& **string**::**back** ()*
const *char*& **string**::**back** () const*

- Обе формы возвращают последний символ строки.
- Вызов функции `back()` для пустой строки приводит к непредсказуемым последствиям.
- При модификации неконстантной строки или перераспределении памяти возвращаемая ссылка может оказаться некорректной (см. раздел 13.2.6).

13.3.6. Создание C-строк и массивов символов

const *char** **string**::**c_str** () const*
const *char** **string**::**data** () const*

- Возвращают содержимое строки в виде массива символов, включая символ `'\0'`, замыкающий строку. Таким образом, результат представляет собой корректную C-строку, полученную из объекта класса `string`.
- Возвращаемое значение является содержимым строки. Таким образом, вызывающая сторона никогда не должна ни модифицировать, ни удалять из памяти это значение.
- Возвращаемое значение является корректным только тогда, когда строка существует или когда к ней применяются только константные функции.
- До принятия стандарта C++11 гарантировалось, что значение, возвращаемое функцией `data()`, содержит все символы строки за исключением замыкающего символа `'\0'`. Таким образом, значение, возвращаемое функцией `data()`, *не было* корректной C-строкой.

size_type **string**::**copy** (*char** *buf*, size_type *buf_size*) const*
size_type **string**::**copy** (*char** *buf*, size_type *buf_size*,
size_type *idx*) const*

- Обе формы копируют не более *buf_size* символов строки (начиная с индекса *idx*, если он передается в виде аргумента) в символьный массив *buf*.
- Функции возвращают количество скопированных символов.
- Нулевой символ не добавляется. Таким образом, содержимое массива *buf* может *не быть* корректной C-строкой после вызова.
- Вызывающая сторона должна гарантировать, что массив *buf* достаточно большой, в противном случае могут возникнуть непредсказуемые последствия.
- Генерирует исключение `out_of_range`, если *idx* > `size()`.

13.3.7. Модифицирующие операции

Присваивания

string& **string::operator** = (const *string*& *str*)
string& **string::assign** (const *string*& *str*)

- Копирующий оператор присваивания.
- Обе операции присваивают значение строки *str*.
- Обе операции возвращают `*this`.

string& **string::operator** = (*string*&& *str*)
string& **string::assign** (*string*&& *str*)

- Перемещающий оператор присваивания.
- Перемещает содержимое строки *str* в `*this`.
- После выполнения операции содержимое строки *str* становится неопределенным.
- Возвращает `*this`.
- Доступно начиная со стандарта C++11.

string& **string::assign** (const *string*& *str*, size_type *str_idx*,
size_type *str_num*)

- Присваивает не более *str_num* символов строки *str*, начиная с индекса *str_idx*.
- Возвращает `*this`.
- Генерирует исключение `out_of_range`, если *str_idx* > *str*.`size()`.

string& **string::operator** = (const *char** *cstr*)
string& **string::assign** (const *char** *cstr*)

- Обе операции присваивают символы С-строки *cstr*.
- Они присваивают все символы С-строки *cstr* за исключением `'\0'`.
- Обе операции возвращают `*this`.
- Передача нулевого указателя (`nullptr` или `NULL`) приводит к непредсказуемым последствиям.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

string& **string::assign** (const *char** *chars*, size_type *chars_len*)

- Присваивает *chars_len* символов из массива символов *chars*.
- Возвращает `*this`.
- Массив *chars* должен содержать не менее *chars_len* символов. Символы могут быть любыми. Таким образом, символ `'\0'` не имеет специального значения.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

string& **string::operator =** (*char c*)

- Присваивает символ *c* как новое значение.
- Возвращает `*this`.
- После вызова строка `*this` содержит только этот единственный символ.

string& **string::assign** (`size_type` *num*, *char c*)

- Присваивает *num* экземпляров символов *c*.
- Возвращает `*this`.
- Генерирует исключение `length_error`, если *num* равно *string*`::npos`.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

string& **string::assign** (`InputIterator` *beg*, `InputIterator` *end*)

- Присваивает все символы диапазона [*beg*,*end*).
- Возвращает `*this`.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

string& **string::operator =** (*initializer-list*)
string& **string::assign** (*initializer-list*)

- Обе операции присваивают символы списка инициализации *initializer-list*.
- Обе операции возвращают `*this`.
- Обе операции генерируют исключение `length_error`, если результирующий размер превышает максимальное количество символов.
- Доступно начиная со стандарта C++11.

`void` **string::swap** (*string*& *str*)
`void` **swap** (*string*& *str1*, *string*& *str2*)

- Обе формы обменивают значения двух строк, `*this` и *str*, а также *str1* и *str2*.
- Эти функции предпочтительнее копирующего присваивания, потому что выполняются быстрее. Фактически они гарантируют константную сложность (см. раздел 13.2.8).

Добавление символов

string& **string::operator +=** (`const` *string*& *str*)
string& **string::append** (`const` *string*& *str*)

- Обе операции добавляют символы строки *str*.
- Они возвращают `*this`.
- Обе операции генерируют исключение `length_error`, если результирующий размер превышает максимальное количество символов.

string& **string::append** (const *string*& *str*, size_type *str_idx*, size_type *str_num*)

- Добавляет не более *str_num* символов *str*, начиная с индекса *str_idx*.
- Возвращает *this.
- Генерирует исключение out_of_range, если *str_idx* > *str*.size().
- Генерирует исключение length_error, если результирующий размер превышает максимальное количество символов.

string& **string::operator +=** (const *char** *cstr*)
string& **string::append** (const *char** *cstr*)

- Обе операции добавляют символы С-строки *cstr*.
- Они возвращают *this.
- Передача нулевого указателя (nullptr или NULL) приводит к непредсказуемым последствиям.
- Обе операции генерируют исключение length_error, если результирующий размер превышает максимальное количество символов.

string& **string::append** (const *char** *chars*, size_type *chars_len*)

- Добавляет *chars_len* символов массива символов *chars*.
- Возвращает *this.
- Массив *chars* должен содержать не менее *chars_len* символов. Символы могут быть любыми. Таким образом, символ '\0' не имеет специального значения.
- Генерирует исключение length_error, если результирующий размер превышает максимальное количество символов.

string& **string::append** (size_type *num*, *char c*)

- Добавляет *num* экземпляров символа *c*.
- Возвращает *this.
- Генерирует исключение length_error, если результирующий размер превышает максимальное количество символов.

string& **string::operator +=** (*char c*)
void **string::push_back** (*char c*)

- Обе операции добавляют символ *c*.
- Оператор += возвращает *this.
- Обе операции генерируют исключение length_error, если результирующий размер превышает максимальное количество символов.

string& **string::append** (InputIterator *beg*, InputIterator *end*)

- Добавляет все символы диапазона [*beg*,*end*).
- Возвращает *this.
- Генерирует исключение length_error, если результирующий размер превышает максимальное количество символов.

string& **string::operator +=** (*initializer-list*)
void **string::append** (*initializer-list*)

- Обе операции добавляют все символы списка инициализации *initializer-list*.
- Обе операции возвращают `*this`.
- Обе операции генерируют `length_error`, если результирующий размер превышает максимальное количество символов.
- Доступно начиная со стандарта C++11.

Вставка символов

string& **string::insert** (`size_type` *idx*, `const` *string*& *str*)

- Вставляет символы строки *str*, так что новые символы начинаются с индекса *idx*.
- Возвращает `*this`.
- Генерирует исключение `out_of_range`, если *idx* > `size()`.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

string& **string::insert** (`size_type` *idx*, `const` *string*& *str*,
`size_type` *str_idx*, `size_type` *str_num*)

- Вставляет не более *str_num* символов строки *str*, начиная с индекса *str_idx*, так что новые символы начинаются с индекса *idx*.
- Возвращает `*this`.
- Генерирует исключение `out_of_range`, если *idx* > `size()`.
- Генерирует исключение `out_of_range`, если *str_idx* > *str*`.size()`.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

string& **string::insert** (`size_type` *idx*, `const` *char** *cstr*)

- Вставляет символы С-строки *cstr*, так что новые символы начинаются с индекса *idx*.
- Возвращает `*this`.
- Передача нулевого указателя (`nullptr` или `NULL`) приводит к непредсказуемым последствиям.
- Генерирует исключение `out_of_range`, если *idx* > `size()`.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

string& **string::insert** (`size_type` *idx*, `const` *char** *chars*,
`size_type` *chars_len*)

- Вставляет *chars_len* символов массива символов *chars*, так что новые символы начинаются с индекса *idx*.
- Возвращает `*this`.
- Массив *chars* должен содержать как минимум *chars_len* символов. Символы могут быть любыми. Таким образом, символ `'\0'` не имеет специального значения.

- Генерирует исключение `out_of_range`, если *idx* > `size()`.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

string& **`string::insert`** (`size_type` *idx*, `size_type` *num*, *char c*)
`iterator` **`string::insert`** (`const_iterator` *pos*, `size_type` *num*, *char c*)

- Вставляет *num* экземпляров символа *c* в позицию, указанную индексом *idx* или итератором *pos* соответственно.
- Первая форма вставляет новые символы так, что они начинаются с индекса *idx*.
- Вторая форма вставляет новые символы перед символом, на который ссылается итератор *pos*.
- Первая форма возвращает `*this`.
- Вторая форма возвращает позицию первого вставленного символа или итератор *pos*, если вставка не была выполнена.
- Перегрузка обеих этих функций может вызвать неоднозначность. Если передать `0` как первый аргумент, он может интерпретироваться либо как индекс, что обычно сопровождается преобразование в тип `unsigned`, либо как итератор, что обычно означает преобразования в тип `char*`. В этом случае следует передавать индекс вместе с его точным типом. Рассмотрим пример:

```
std::string s;
...
s.insert(0,1,' ');                              // ОШИБКА: неоднозначность
s.insert((std::string::size_type)0,1,' ');      // OK
```

- Обе формы генерируют исключение `out_of_range`, если *idx* > `size()`.
- Обе формы генерируют исключение `length_error`, если результирующий размер превышает максимальное количество символов.
- До принятия стандарта C++11 аргумент *pos* имел тип `iterator`, а значение, возвращаемое второй формой, имело тип `void`.

`iterator` **`string::insert`** (`const_iterator` *pos*, *char c*)

- Вставляет копию символа *c* перед символом, на который ссылается итератор *pos*.
- Возвращает позицию вставленного символа.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.
- До принятия стандарта C++11 аргумент *pos* имел тип `iterator`.

`iterator` **`string::insert`** (`const_iterator` *pos*, `InputIterator` *beg*, `InputIterator` *end*)

- Вставляет все символы диапазона [*beg*,*end*) *c* перед символом, на который ссылается итератор *pos*.

- Вторая форма возвращает позицию первого вставленного символа или итератор *pos*, если вставка не была выполнена.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.
- До принятия стандарта C++11 аргумент *pos* имел тип `iterator`, а значение, возвращаемое функцией, имело тип `void`.

`iterator` ***`string`*`::insert`** `(const_iterator` *pos, initializer-list*`)`

- Вставляет все символы списка инициализации *initializer-list* перед символом, на который ссылается итератор *pos*.
- Возвращает позицию первого вставленного символа или итератор *pos*, если вставка не была выполнена.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

Удаление символов

`void` ***`string`*`::clear`** `()`
string`&` ***`string`*`::erase`** `()`

- Обе функции удаляют все символы строки. Таким образом, строка после вызова становится пустой.
- Функция `erase()` возвращает `*this`.

string`&` ***`string`*`::erase`** `(size_type` *idx*`)`
string`&` ***`string`*`::erase`** `(size_type` *idx*`, size_type` *len*`)`

- Обе формы удаляют не более *len* символов строки `*this`, начиная с индекса *idx*.
- Обе формы возвращают `*this`.
- Если аргумент *len* пропущен, то удаляются все оставшиеся символы.
- Обе формы генерируют исключение `out_of_range`, если *idx* > `size()`.

`iterator` ***`string`*`::erase`** `(const_iterator` *pos*`)`
`iterator` ***`string`*`::erase`** `(const_iterator` *beg*`,`
`const_iterator` *end*`)`

- Обе формы удаляют отдельный символ в позиции итератора *pos* или все символы диапазона [*beg*,*end*) соответственно.
- Обе формы возвращают позицию первого символа, стоящего после последнего удаленного символа (таким образом, вторая форма возвращает *end*).
- До принятия стандарта C++11 аргументы *pos*, *beg* и *end* имели тип `iterator`.

`void` ***`string`*`::pop_back`** `()`

- Стирает последний символ.
- Вызов этой функции для пустой строки приводит к непредсказуемым последствиям.
- Доступно начиная со стандарта C++11.

Изменение размера

```
void string::resize (size_type num)
void string::resize (size_type num, char c)
```

- Обе формы изменяют количество символов строки `*this` на *num*. Таким образом, если *num* не равно `size()`, функции добавляют или удаляют символы в конце строки в соответствии с новым размером.
- Если количество символов увеличивается, то новые символы инициализируются символом *c*. Если символ *c* пропущен, то символы инициализируются конструктором по умолчанию символьного типа (для `string` этим символом является `'\0'`).
- Обе формы генерируют исключение `length_error`, если *num* равен *string*`::npos`.
- Обе формы генерируют исключение `length_error`, если результирующий размер превышает максимальное количество символов.

Замена символов

```
string& string::replace (size_type idx, size_type len,
                         const string& str)
string& string::replace (begin_iterator beg,
                         begin_iterator end,
                         const string& str)
```

- Первая форма заменяет не более *len* символов строки `*this`, начиная с индекса *idx*, всеми символами строки *str*.
- Вторая форма заменяет все символы диапазона [*beg*,*end*) всеми символами строки *str*.
- Обе формы возвращают `*this`.
- Обе формы генерируют исключение `out_of_range`, если *idx* > `size()`.
- Обе формы генерируют исключение `length_error`, если результирующий размер превышает максимальное количество символов.
- До принятия стандарта C++11 аргументы *beg* и *end* имели тип `iterator`.

```
string& string::replace (size_type idx, size_type len,
                         const string& str, size_type str_idx,
                         size_type str_num)
```

- Заменяет не более *len* символов строки `*this`, начиная с индекса *idx*, не более чем *str_num* символами строки *str*, начиная с индекса *str_idx*.
- Возвращает `*this`.
- Генерирует исключение `out_of_range`, если *idx* > `size()`.
- Генерирует исключение `out_of_range`, если *str_idx* > *str*`.size()`.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.

```
string& string::replace (size_type idx, size_type len,
                         const char* cstr)
```

```
string& string::replace (const_iterator beg,
                         const_iterator end,
                         const char* cstr)
```

- Обе формы заменяют не более *len* символов строки `*this`, начиная с индекса *idx*, или все символы диапазона [*beg*,*end*) соответственно всеми символами С-строки *cstr*.
- Обе формы возвращают `*this`.
- Передача нулевого указателя (`nullptr` или `NULL`) приводит к непредсказуемым последствиям.
- Обе формы генерируют исключение `out_of_range`, если *idx* > `size()`.
- Обе формы генерируют исключение `length_error`, если результирующий размер превышает максимальное количество символов.
- До принятия стандарта C++11 аргументы *beg* и *end* имели тип `iterator`.

```
string& string::replace (size_type idx, size_type len,
                         const char* chars,
                         size_type chars_len)
string& string::replace (const_iterator beg,
                         const_iterator end,
                         const char* chars,
                         size_type chars_len)
```

- Обе формы заменяют не более *len* символов строки `*this`, начиная с индекса *idx*, или все символы диапазона [*beg*,*end*) соответственно *chars_len* символами массива символов *chars*.
- Обе формы возвращают `*this`.
- Массив *chars* должен содержать как минимум *chars_len* символов. Символы могут быть любыми. Таким образом, символ `'\0'` не имеет специального значения.
- Обе формы генерируют исключение `out_of_range`, если *idx* > `size()`.
- Обе формы генерируют исключение `length_error`, если результирующий размер превышает максимальное количество символов.
- До принятия стандарта C++11 аргументы *beg* и *end* имели тип `iterator`.

```
string& string::replace (size_type idx, size_type len,
                         size_type num, char c)
string& string::replace (const_iterator beg,
                         const_iterator end,
                         size_type num, char c)
```

- Обе формы заменяют не более *len* символов строки `*this`, начиная с индекса *idx*, или все символы диапазона [*beg*,*end*) соответственно *num* экземплярами символа *c*.
- Обе формы возвращают `*this`.
- Обе формы генерируют исключение `out_of_range`, если *idx* > `size()`.
- Обе формы генерируют исключение `length_error`, если результирующий размер превышает максимальное количество символов.
- До принятия стандарта C++11 аргументы *beg* и *end* имели тип `iterator`.

```
string& string::replace (const_iterator beg,
                         const_iterator end,
                         InputIterator newBeg,
                         InputIterator newEnd)
```

- Заменяет все символы диапазона [*beg*,*end*) всеми символами диапазона [*newBeg*,*newEnd*).
- Возвращает `*this`.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.
- До принятия стандарта C++11 аргументы *beg* и *end* имели тип `iterator`.

```
string& string::replace (const_iterator beg,
                         const_iterator end, initializer-list)
```

- Заменяет все символы диапазона [*beg*,*end*) символами списка инициализации *initializer-list*.
- Возвращает `*this`.
- Генерирует исключение `length_error`, если результирующий размер превышает максимальное количество символов.
- Доступно начиная со стандарта C++11.

13.3.8. Поиск

Поиск символа

```
size_type string::find (char c) const
size_type string::find (char c, size_type idx) const
size_type string::rfind (char c) const
size_type string::rfind (char c, size_type maxIdx) const
```

- Эти функции ищут первый/последний символ *c* (начиная с индекса *idx*/*maxIdx*).
- Функция `find()` выполняет прямой поиск и возвращает первое вхождение символа.
- Функция `rfind()` выполняет обратный поиск и возвращает последнее вхождение символа.
- Эти функции возвращают индекс символа в случае успеха и значение *string*`::npos`, если поиск завершился неудачно.

Поиск подстроки

```
size_type string::find (const string& str) const
size_type string::find (const string& str, size_type idx) const
size_type string::rfind (const string& str) const
size_type string::rfind (const string& str, size_type maxIdx) const
```

- Эти функции ищут первую/последнюю подстроку *str*, начиная с индекса *idx*/*maxIdx*.
- Функция `find()` выполняет прямой поиск и возвращает первое вхождение подстроки.

- Функция `rfind()` выполняет обратный поиск и возвращает последнее вхождение подстроки.
- Эти функции возвращают индекс первого символа подстроки в случае успеха и значение *string*`::npos`, если поиск завершился неудачно.

```
size_type string::find (const char* cstr) const
size_type string::find (const char* cstr, size_type idx) const
size_type string::rfind (const char* cstr) const
size_type string::rfind (const char* cstr, size_type maxIdx) const
```

- Эти функции ищут первую/последнюю подстроку, совпадающую с символами С-строки *cstr*, начиная с индекса *idx/maxIdx*.
- Функции `find()` выполняют прямой поиск и возвращают первую подстроку.
- Функции `rfind()` выполняют обратный поиск и возвращают последнюю подстроку.
- Эти функции возвращают индекс первого символа подстроки в случае успеха и значение *string*`::npos`, если поиск завершился неудачно.
- Передача нулевого указателя (`nullptr` или `NULL`) приводит к непредсказуемым последствиям.

```
size_type string::find (const char* chars, size_type idx,
                        size_type chars_len) const
size_type string::rfind (const char* chars, size_type maxIdx,
                         size_type chars_len) const
```

- Эти функции ищут первую/последнюю подстроку, совпадающую с *chars_len* символами массива символов *chars*, начиная с индекса *idx/maxIdx*.
- Функции `find()` выполняют прямой поиск и возвращают первую подстроку.
- Функции `rfind()` выполняют обратный поиск и возвращают последнюю подстроку.
- Эти функции возвращают индекс первого символа подстроки в случае успеха и значение *string*`::npos`, если поиск завершился неудачно.
- Массив *chars* должен содержать как минимум *chars_len* символов. Символы могут быть любыми. Таким образом, символ `'\0'` не имеет специального значения.

Поиск первого из различных символов

```
size_type string::find_first_of (const string& str) const
size_type string::find_first_of (const string& str,
                                 size_type idx) const
size_type string::find_first_not_of (const string& str) const
size_type string::find_first_not_of (const string& str,
                                     size_type idx) const
```

- Эти функции ищут первый символ, который входит (или, соответственно, не входит) в строку *str*, начиная с индекса *idx*.
- Эти функции возвращают индекс символа или подстроки в случае успеха и значение *string*`::npos`, если поиск завершился неудачно.

```
size_type string::find_first_of (const char* cstr) const
size_type string::find_first_of (const char* cstr,
                                 size_type idx) const
size_type string::find_first_not_of (const char* cstr) const
size_type string::find_first_not_of (const char* cstr,
                                     size_type idx) const
```

- Эти функции ищут первый символ, который входит (или, соответственно, не входит) в С-строку *cstr*, начиная с индекса *idx*.
- Эти функции возвращают индекс символа или подстроки в случае успеха и значение *string*::npos, если поиск завершился неудачно.
- Передача нулевого указателя (nullptr или NULL) приводит к непредсказуемым последствиям.

```
size_type string::find_first_of (const char* chars,
                                 size_type idx,
                                 size_type chars_len) const
size_type string::find_first_not_of (const char* chars,
                                     size_type idx,
                                     size_type chars_len) const
```

- Эти функции ищут первый символ, который встречается (или не встречается) среди элементов *chars_len* символов массива символов *chars*, начиная с индекса *idx*.
- Эти функции возвращают индекс символа или подстроки в случае успеха и значение *string*::npos, если поиск завершился неудачно.
- Массив *chars* должен содержать как минимум *chars_len* символов. Символы могут быть любыми. Таким образом, символ '\0' не имеет специального значения.

```
size_type string::find_first_of (char c) const
size_type string::find_first_of (char c, size_type idx) const
size_type string::find_first_not_of (char c) const
size_type string::find_first_not_of (char c, size_type idx) const
```

- Эти функции ищут первый символ, имеющий (или, соответственно, не имеющий) значение *c* (начиная с индекса *idx*).
- Эти функции возвращают индекс символа или подстроки в случае успеха и значение *string*::npos, если поиск завершился неудачно.

Поиск последнего отличающегося символа

```
size_type string::find_last_of (const string& str) const
size_type string::find_last_of (const string& str, size_type maxIdx) const
size_type string::find_last_not_of (const string& str) const
size_type string::find_last_not_of (const string& str, size_type maxIdx)
                                    const
```

- Эти функции ищут последний символ, являющийся (или не являющийся) элементом строки *str* (выполняя поиск в обратном направлении, начиная с индекса *maxIdx*).

- Эти функции возвращают индекс символа или подстроки в случае успеха и значение *string*::npos, если поиск завершился неудачно.

```
size_type string::find_last_of (const char* cstr) const
size_type string::find_last_of (const char* cstr, size_type maxIdx) const
size_type string::find_last_not_of (const char* cstr) const
size_type string::find_last_not_of (const char* cstr, size_type maxIdx)
                                    const
```

- Эти функции ищут последний символ, являющийся (или не являющийся) элементом С-строки *cstr* (выполняя поиск в обратном направлении, начиная с индекса *maxIdx*).
- Эти функции возвращают индекс символа или подстроки в случае успеха и значение *string*::npos, если поиск завершился неудачно.
- Передача нулевого указателя (nullptr или NULL) приводит к непредсказуемым последствиям.

```
size_type string::find_last_of (const char* chars, size_type maxIdx,
                                size_type chars_len) const
size_type string::find_last_not_of (const char* chars, size_type maxIdx,
                                    size_type chars_len) const
```

- Эти функции ищут последний символ, встречающийся (или не встречающийся) среди *chars_len* символов массива символов *chars* (выполняя поиск в обратном направлении, начиная с индекса *maxIdx*).
- Эти функции возвращают индекс символа или подстроки в случае успеха и значение *string*::npos, если поиск завершился неудачно.
- Массив *chars* должен содержать как минимум *chars_len* символов. Символы могут быть любыми. Таким образом, символ '\0' не имеет специального значения.

```
size_type string::find_last_of (char c) const
size_type string::find_last_of (char c, size_type maxIdx) const
size_type string::find_last_not_of (char c) const
size_type string::find_last_not_of (char c, size_type maxIdx) const
```

- Эти функции ищут последний символ, имеющий (или не имеющий) значение *c* соответственно (выполняя поиск в обратном направлении, начиная с индекса *maxIdx*).
- Эти функции возвращают индекс символа или подстроки в случае успеха и значение *string*::npos, если поиск завершился неудачно.

13.3.9. Подстроки и конкатенация строк

```
string string::substr () const
string string::substr (size_type idx) const
string string::substr (size_type idx, size_type len) const
```

- Все формы возвращают подстроку, состоящую из не более чем *len* символов строки *this, начиная с индекса *idx*.
- Если аргумент *len* пропущен, используются все оставшиеся символы.
- Если аргументы *idx* и *len* пропущены, возвращается копия строки.
- Все формы генерируют исключение out_of_range, если *idx* > size().

string **operator +** (const *string*& *str1*, const *string*& *str2*)
string **operator +** (*string*&& *str1*, *string*&& *str2*)
string **operator +** (*string*&& *str1*, const *string*& *str2*)
string **operator +** (const *string*& *str1*, *string*&& *str2*)
string **operator +** (const *string*& *str*, const *char** *cstr*)
string **operator +** (*string*&& *str*, const *char** *cstr*)
string **operator +** (const *char** *cstr*, const *string*& *str*)
string **operator +** (const *char** *cstr*, *string*&& *str*)
string **operator +** (const *string*& *str*, *char c*)
string **operator +** (*string*&& *str*, *char c*)
string **operator +** (*char c*, const *string*& *str*)
string **operator +** (*char c*, *string*&& *str*)

- Все формы выполняют конкатенацию символов, принадлежащих обоим операндам, и возвращают суммарную строку.
- Если аргумент является rvalue-ссылкой, то используется семантика перемещения. Это значит, что после выполнения вызова аргумент имеет неопределенное значение.
- Операндом может быть:
  - строка;
  - С-строка;
  - отдельный символ.
- Все формы генерируют исключение `length_error`, если результирующий размер превышает максимальное количество символов.

13.3.10. Функции ввода-вывода

ostream& **operator <<** (*ostream*&& *strm*, const *string*& *str*)

- Записывает символы строки *str* в поток *strm*.
- Если *strm*.`width()` больше 0, то записываются не более `width()` символов, и значение `width()` устанавливается равным 0.
- Аргумент *ostream* имеет тип потока вывода `basic_ostream<`*char*`>`, соответствующий символьному типу (подробнее об этом — в разделе 15.2.1).
- До принятия стандарта C++11 тип потока представлял собой lvalue-ссылку.

istream& **operator >>** (*istream*&& *strm*, *string*& *str*)

- Читает символы следующего слова из потока *strm* в строку *str*.
- Если для потока *strm* установлен флаг `skipws`, начальные пробельные символы игнорируются.
- Символы извлекаются, пока не произойдет одно из следующих событий:
  - значение *strm*.`width()` больше 0, и в строку помещено `width()` символов;
  - значение *strm*.`good()` равно `false` (в результате может генерироваться соответствующее исключение);
  - для следующего символа `c` значение `isspace(c,strm.getloc())` равно `true`;
  - в строке хранятся *str*.`max_size()` символов.

- Внутренняя память перераспределяется соответствующим образом.
- Аргумент *istream* имеет тип потока ввода `basic_istream<`*char*`>` в соответствии с символьным типом (см. раздел 15.2.1).
- До принятия стандарта C++11 тип потока представлял собой lvalue-ссылку.

istream& **getline** (*istream*& *strm*, *string*& *str*)
istream& **getline** (*istream*&& *strm*, *string*& *str*)
istream& **getline** (*istream*& *strm*, *string*& *str*, *char delim*)
istream& **getline** (*istream*&& *strm*, *string*& *str*, *char delim*)

- Читает символы следующей строки из потока *strm* в строку *str*.
- Все символы, включая начальные пробельные символы, извлекаются, пока не произойдет одно из следующих событий:
  - значение *strm*`.good()` равно `false` (в результате может сгенерироваться соответствующее исключение);
  - из потока извлекаются аргументы *delim* или *strm*`.widen('\n');`
  - в строке хранятся *str*`.max_size()` символов.
- Разделитель строк извлекается, но не добавляется.
- Внутренняя память перераспределяется соответствующим образом.
- Аргумент *istream* имеет тип потока ввода `basic_istream<`*char*`>` в соответствии с символьным типом (см. раздел 15.2.1).
- Перегрузки для rvalue-ссылок стали доступными после принятия стандарта C++11.

13.3.11. Числовые преобразования

```
int stoi (const string& str, size_t* idxRet = nullptr, int base = 10)
long stol (const string& str, size_t* idxRet = nullptr, int base = 10)
unsigned long stoul (const string& str, size_t* idxRet = nullptr,
                     int base = 10)
long long stoll (const string& str, size_t* idxRet = nullptr, int base = 10)
unsigned long long stoull (const string& str, size_t* idxRet = nullptr,
                           int base = 10)
float stof (const string& str, size_t* idxRet = nullptr, int base = 10)
double stod (const string& str, size_t* idxRet = nullptr, int base = 10)
long double stold (const string& str, size_t* idxRet = nullptr, int base = 10)
```

- Преобразует строку *str* в соответствующий тип.
- Строка *str* может иметь тип `string` или `wstring`.
- Начальные пробельные символы игнорируются.
- Если *idxRet*`!=nullptr`, то функции возвращают индекс первого символа, который не подвергся преобразованию.
- Аргумент `base` позволяет задать основу системы счисления.

- Функции могут генерировать исключение `std::invalid_argument`, если преобразование невозможно, и `std::out_of_range`, если преобразованное значение выходит за допустимые пределы, установленные для возвращаемых значений.

```
string to_string (Type val)
wstring to_wstring (Type val)
```

- Преобразует *val* в объект класса `string` или `wstring`.
- Корректными типами для значения *val* являются `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, `double` или `long double`.

13.3.12. Генерация итераторов

```
iterator string::begin ()
const_iterator string::begin () const
const_iterator string::cbegin ()
```

- Все формы возвращают итератор произвольного доступа для начала строки (позицию первого символа).
- Если строка пустая, то вызов эквивалентен вызовам функции `end()` или `cend()`.

```
iterator string::end ()
const_iterator string::end () const
const_iterator string::cend ()
```

- Все формы возвращают итератор произвольного доступа для конца строки (позицию после последнего символа).
- Символ в конце строки не определен. Таким образом, вызовы `*s.end()` и `*s.cend()` могут привести к непредсказуемым последствиям.
- Если строка пустая, то вызов эквивалентен вызовам функции `begin()` или `cbegin()`.

```
reverse_iterator string::rbegin ()
const_reverse_iterator string::rbegin () const
const_reverse_iterator string::crbegin ()
```

- Все формы возвращают итератор произвольного доступа для начала строки при обратном обходе (позицию последнего символа).
- Если строка пустая, то вызов эквивалентен вызовам функции `rend()` или `crend()`.
- Подробное описание обратных итераторов приведено в разделе 9.4.1.

```
reverse_iterator string::rend ()
const_reverse_iterator string::rend () const
const_reverse_iterator string::crend ()
```

- Все формы возвращают итератор произвольного доступа для конца строки при обратном обходе (позицию перед первым символом).
- Символ конца строки при обратном обходе не определен. Таким образом, вызовы `*s.rend()` и `*s.crend()` могут привести к непредсказуемым последствиям.

- Если строка пустая, то вызов эквивалентен вызовам функций `rbegin()` или `crbegin()`.
- Подробное описание обратных итераторов приведено в разделе 9.4.1.

13.3.13. Поддержка механизмов распределения памяти

Строковые классы имеют обычные функции-члены, предназначенные для поддержки механизмов распределения памяти.

***string*::allocator_type**

- Тип механизма распределения памяти.
- Третий шаблонный параметр класса `basic_string<>`.
- Для типа `string` эквивалент `allocator<char>`.

`allocator_type` ***string*::get_allocator** `() const`

- Возвращает модель памяти для строки.

Строковые классы имеют все конструкторы с необязательными аргументами, задающими механизм распределения памяти. Ниже перечислены все строковые конструкторы, включая необязательные аргументы для механизмов распределения памяти в соответствии со стандартом[12].

```
namespace std {
   template <typename charT,
             typename traits = char_traits<charT>,
             typename Allocator = allocator<charT> >
   class basic_string {
      public:
      // конструктор по умолчанию
      explicit basic_string(const Allocator& a = Allocator());

      // копирующий и перемещающий конструкторы
      //(с механизмом распределения памяти)
      basic_string(const basic_string& str);
      basic_string(basic_string&& str);
      basic_string(const basic_string& str, const Allocator&);
      basic_string(basic_string&& str, const Allocator&);

      // конструктор для подстрок
     basic_string(const basic_string& str,
                  size_type str_idx = 0,
                  size_type str_num = npos,
                  const Allocator& a = Allocator());

     // конструктор для С-строк
     basic_string(const charT* cstr,
                  const Allocator& a = Allocator());

     // конструктор для символьных массивов
```

[12] Копирующий конструктор с механизмом распределения памяти, перемещающий конструктор и конструктор для списка инициализации, доступны начиная со стандарта C++11.

```
    basic_string(const charT* chars, size_type chars_len,
                 const Allocator& a = Allocator());

    // конструктор для num экземпляров символа
    basic_string(size_type num, charT c,
                 const Allocator& a = Allocator());

    // конструктор диапазона символов
    template <typename InputIterator>
    basic_string(InputIterator beg, InputIterator end,
                 const Allocator& a = Allocator());

    // конструктор для списка инициализации
    basic_string(initializer_list<charT>,
                 const Allocator& a = Allocator());
       ...
    };
}
```

Эти конструкторы описаны в разделе 13.3.2. Там же перечислены дополнительные возможности, которые можно передать с помощью собственного объекта модели памяти. Если строка инициализируется другой строкой, то копируется также механизм распределения памяти[13]. Подробное описание механизмов распределения памяти приведено в главе 19.

[13] В исходном варианте стандарта указано, что при копировании строк используется механизм распределения памяти по умолчанию. Однако в этом не было большого смысла, поэтому было предложено исправить это решение.

Глава 14

Регулярные выражения

В главе описывается библиотека регулярных выражений, позволяющая использовать метасимволы (wildcards) и шаблоны (patterns) для поиска и замен символов в строках.

С помощью регулярных выражений можно выполнять ряд операций.

- **Сравнивать** входную строку с регулярным выражением.
- **Искать** шаблоны, соответствующие регулярному выражению.
- **Размечать** символы в соответствии с разделителем токенов, заданным в виде регулярного выражения.
- **Заменять** первую или все подпоследовательности, соответствующие регулярному выражению.

Для всех этих операций можно использовать разные грамматики, которые используются для определения регулярного выражения.

Глава начинается с краткого перечисления разных операций, затем обсуждаются разные грамматики, а в заключение приводятся подробные описания операций над регулярными выражениями.

14.1. Интерфейс сравнения и поиска регулярных выражений

Рассмотрим, как можно проверить, соответствует ли последовательность символов полностью или частично конкретному регулярному выражению:

```
// regex/regex1.cpp

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

void out (bool b)
{
   cout << ( b ? "found" : "not found") << endl;
}

int main()
{
   // ищем значение с дескрипторами XML/HTML (используя синтаксис по умолчанию):
   regex reg1("<.*>.*</.*>");
   bool found = regex_match ("<tag>value</tag>", // данные
                             reg1);              // регулярное выражение
```

```
    out(found);

    // ищем значение с дескрипторами XML/HTML
    // (дескрипторы до и после значения должны совпадать):
    regex reg2("<(.*)>.*</\\1>");
    found = regex_match ("<tag>value</tag>",       // данные
                         reg2);                    // регулярное выражение
    out(found);

    // ищем значение с дескрипторами XML/HTML (используя синтаксис команды grep):
    regex reg3("<\\(.*\\)>.*</\\1>",regex_constants::grep);
               found = regex_match ("<tag>value</tag>", // данные
                                    reg3);              // регулярное выражение
    out(found);

    // используем С-строку как регулярное выражение
    // (требуется явное приведение к regex):
    found = regex_match ("<tag>value</tag>",        // данные
                         regex("<(.*)>.*</\\1>"));  // регулярное выражение
    out(found);
    cout << endl;

    // сравнение regex_match() и regex_search():
    found = regex_match ("XML tag: <tag>value</tag>",
                         regex("<(.*)>.*</\\1>"));      // нет соответствия
    out(found);

    found = regex_match ("XML tag: <tag>value</tag>",
                         regex(".*<(.*)>.*</\\1>.*"));  // есть соответствие

    out(found);

    found = regex_search ("XML tag: <tag>value</tag>",
                          regex("<(.*)>.*</\\1>"));     // есть соответствие
    out(found);

    found = regex_search ("XML tag: <tag>value</tag>",
                          regex(".*<(.*)>.*</\\1>.*")); // есть соответствие
    out(found);
}
```

Сначала мы подставляем необходимый заголовочный файл и глобальные идентификаторы пространства имен `std`:

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

Затем показано, как определить регулярное выражение для проверки соответствия символьной последовательности конкретному шаблону. Мы объявляем и инициализируем объект `reg1` как регулярное выражение:

```
regex reg1("<.*>.*</.*>");
```

Объект, представляющий регулярное выражение, имеет тип `std::regex`. Как и при работе со строками, этот тип представляет собой специализацию класса

`std::basic_regex<>` для символьного типа `char`. Для символьного типа `wchar_t` предусмотрен класс `std::wregex`.

Объект `reg1` инициализируется регулярным выражением

```
<.*>.*</.*>
```

Эти регулярные выражения проверяют строку на соответствие шаблону "*<символы>символы</символы>*", используя синтаксис `.*`, где символ "`.`" означает "любой символ за исключением символа новой строки", а символ "`*`" означает "ни разу или несколько раз". Таким образом, мы пытаемся проверять соответствие формату XML- или HTML-дескрипторов. Последовательность символов `<tag>value</tag>` соответствует этому шаблону, поэтому выражение

```
regex_match ("<tag>value</tag>", // данные

             reg1);              // регулярное выражение
```

равно `true`.

Можно даже указать, что начальные и заключительные дескрипторы должны быть одинаковыми символьными последовательностями, что демонстрируют следующие операторы:

```
regex reg2("<(.*)>.*</\\1>");
          found = regex_match ("<tag>value</tag>", // данные
                               reg2);              // регулярное выражение
```

Выражение `regex_match()` снова равно `true`.

Здесь используется концепция группировки. Мы используем конструкцию "`(...)`" для определения так называемой *группы захвата* (capture group), на которую впоследствии ссылаемся с помощью регулярного выражения "`\1`". Однако мы задаем регулярное выражение как обычную символьную последовательность, поэтому должны определить "символ \, за которым следует символ 1", как "`\\1`". В качестве альтернативы можно было бы использовать *неформатированную строку* (raw string), введенную в стандарте C++11 (см. раздел 3.1.6):

```
R"(<(.*)>.*</\1>)" // эквивалент: "<(.*)>.*</\\1>"
```

Такая неформатированная строка позволяет определить символьную последовательность, написав ее точное содержание как неформатированную последовательность символов. Она начинается с символов "`R"(`" и заканчивается символами "`)"`". Для того чтобы иметь возможность использовать символы "`)"`" в неформатированной строке, можно использовать разделитель. Таким образом, полный синтаксис неформатированной строки имеет вид `R"`*delim*`(...)`*delim*`"`, где *delim* — последовательность символов, состоящих не более чем из 16 основных символов, за исключением обратной косой черты, пробелов и скобок.

Здесь мы вводим специальные символы для регулярных выражений как часть грамматики. Стандартная библиотека C++ поддерживает разные грамматики. По умолчанию используется грамматика "модифицированная грамматика ECMAScript," подробно описанная в разделе 14.8. Следующие строки демонстрируют использование другой грамматики:

```
regex reg3("<\\(.*\\)>.*</\\1>",regex_constants::grep);
found = regex_match ("<tag>value</tag>", // данные
                     reg3);              // регулярное выражение
```

Здесь необязательный второй аргумент `regex_constants::grep` конструктора `regex` задает грамматику, напоминающую команду UNIX `grep`, где, например, необходимо задавать маску группирования символов с помощью дополнительных обратных косых черт (которые в обычных строковых литералах необходимо дополнять обратными косыми чертами). Различия между разными грамматиками рассматривается в разделе 14.9.

Для регулярного выражения все предыдущие примеры использовали отдельный объект. Это не обязательно, однако простой передачи строки или строкового литерала в виде регулярного выражения недостаточно. Хотя в программе объявлено неявное преобразование типа, итоговое выражение не компилируется, потому что оно неоднозначно. Рассмотрим пример:

```
regex_match ("<tag>value</tag>",          // ОШИБКА: неоднозначность
             "<(.*)>.*</\\1>")
regex_match (string("<tag>value</tag>"), // ОШИБКА: неоднозначность
             "<(.*)>.*</\\1>")
regex_match ("<tag>value</tag>",          // ОК
             regex("<(.*)>.*</\\1>"))
```

В заключение рассмотрим различия между функциями `regex_match()` и `regex_search()`:

- функция `regex_match()` проверяет, соответствует ли символьная последовательность регулярному выражением *полностью*;
- функция `regex_search()` проверяет, соответствует ли символьная последовательность регулярному выражению *частично*.

Других различий нет. Таким образом, оператор

```
regex_search (data, regex(pattern))
```

всегда эквивалентен оператору

```
regex_match (data, regex("(.|\n)*"+pattern+"(.|\n)*"))
```

где символы "`(.|\n)*`" означают любое количество любых символов (символ "`.`" означает любой символ, за исключением символа перехода на новую строку, а символ "`|`" означает "или").

Читатели могли бы возразить, что эти выражения не дают важную информацию, по крайней мере в случае функции `regex_search()`: *где именно* символьная последовательность соответствует регулярному выражению. Для реализации этой и многих других возможностей были введены новые версии функций `regex_match()` и `regex_search()`, в которых новый параметр возвращает всю необходимую информацию о соответствии.

14.2. Работа с подвыражениями

Рассмотрим следующий пример:

```
// regex/regex2.cpp

#include <string>
#include <regex>
#include <iostream>
#include <iomanip>
```

```
using namespace std;

int main()
{
   string data = "XML tag: <tag-name>the value</tag-name>.";
   cout << "data: " << data << "\n\n";

   smatch m; // для возвращаемой информации о соответствии
   bool found = regex_search (data,
                              m,
                              regex("<(.*)>(.*)</(\\1)>"));

   // выводим детали соответствий
   cout << "m.empty():        " << boolalpha << m.empty() << endl;
   cout << "m.size():         " << m.size() << endl;
   if (found) {
       cout << "m.str():          " << m.str() << endl;
       cout << "m.length():       " << m.length() << endl;
       cout << "m.position():     " << m.position() << endl;
       cout << "m.prefix().str(): " << m.prefix().str() << endl;
       cout << "m.suffix().str(): " << m.suffix().str() << endl;
       cout << endl;

      // перебор всех соответствий (с помощью индекса соответствий)
      for (int i=0; i<m.size(); ++i) {
         cout << "m[" << i << "].str():    " << m[i].str() << endl;
         cout << "m.str(" << i << "):      "  << m.str(i) << endl;
         cout << "m.position(" << i << "): " << m.position(i)
              << endl;
      }
      cout << endl;

      // перебор всех соответствий (с помощью итератора)
      cout << "matches:" << endl;
      for (auto pos = m.begin(); pos != m.end(); ++pos) {
           cout << " " << *pos << " ";
           cout << "(length: " << pos->length() << ")" << endl;
      }
   }
}
```

В этом примере демонстрируется использование объектов `match_results`, которые можно передать функциям `regex_match()` и `regex_search()` для получения деталей соответствий. Класс `std::match_results<>` — это шаблон, который должен конкретизироваться типом итератора для обрабатываемых символов. Стандартная библиотека C++ содержит несколько заранее определенных конкретизаций:

- `smatch`: для деталей соответствий в строках;
- `cmatch`: для деталей соответствий в C-строках (`const char*`);
- `wsmatch`: для деталей соответствий в объектах класса `wstrings`;
- `wcmatch`: для деталей соответствий в широких C-строках (`const wchar_t*`).

Таким образом, если применить функцию `regex_match()` или `regex_search()` к строкам C++, то следует использовать тип `smatch`, а для обычных строковых литералов должен использоваться тип `cmatch`.

В примере показано, что выводит объект `match_results`, который выполняет поиск регулярного выражения

```
<(.*)>(.*)</(\1)>
```

в строке `data`, инициализированной следующей символьной строкой:

```
"XMLtag: <tag-name>the value</tag-name>
```

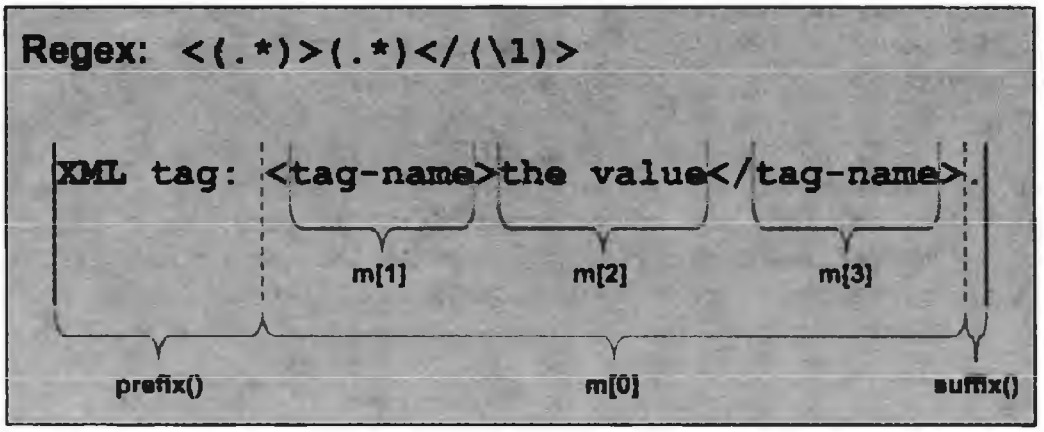


Рис. 14.1. Интерфейс соответствия регулярного выражения

После вызова объект `m` типа `match_results` имеет состояние, которое показано на рис. 14.1 и обеспечивает следующий интерфейс.

- Объект класса `match_results` содержит:
  - объект `m[0]` типа `sub_match` для всех соответствующих символов;
  - объект `prefix()` типа `sub_match`, представляющий все символы, предшествующие первому соответствующему символу;
  - объект `suffix()` типа `sub_match`, представляющий все символы, следующие после последнего соответствующего символа.
- Кроме того, любая "группа захвата" имеет доступ к соответствующему объекту `m[n]` типа `sub_match`. Поскольку в данном примере регулярное выражение определяет три "группы захвата" (одна для начального дескриптора, одна для значения и одна для заключительного дескриптора), они доступны как `m[1]`, `m[2]` и `m[3]`.
- Функция `size()` возвращает количество объектов типа `sub_match` (включая `m[0]`).
- Все объекты типа `sub_match` порождены из класса `pair<>` и содержат первый символ как член `first`, и позицию, следующую за последним символом, как член `second`. Кроме того, функция `str()` возвращает символы как строку, функция `length()` возвращает количество символов, оператор `<<` записывает символы в поток, а кроме того, для них определено неявное преобразование в строку.
- Кроме того, объект класса `match_results` содержит:

- функцию-член `str()` для создания строки в целом (с помощью вызова `str()` или `str(0)`) или *n*-й соответствующей подстроки (с помощью вызова `str(n)`), которая пуста, если соответствующих подстрок не существует (таким образом, допускается передача параметра *n*, который превышает `size()`);
- функцию-член `length()` для вычисления длины соответствующей строки в целом (с помощью вызова `length()` или `length(0)`) или длины *n*-й соответствующей подстроки (с помощью вызова `length(n)`), которая равна 0, если соответствующих подстрок не существует (таким образом, допускается передача параметра *n*, который превышает `size()`);
- функцию-член `position()` для вычисления позиции соответствующей подстроки в целом (с помощью вызова `position()` или `position(0)`) или позиции *n*-й соответствующей подстроки (с помощью вызова `length(n)`);
- функции-члены `begin()`, `cbegin()`, `end()` и `cend()` для перебора объектов от `m[0]` до `m[n]` типа `sub_match`;

Программа выводит следующий результат:

```
data:               XML tag: <tag-name>the value</tag-name>.

m.empty():          false
m.size():           4
m.str():            <tag-name>the value</tag-name>
m.length():         30
m.position():       9
m.prefix().str():   XML tag:
m.suffix().str():   .
m[0].str():         <tag-name>the value</tag-name>
m.str(0):           <tag-name>the value</tag-name>
m.position(0):      9
m[1].str():         tag-name
m.str(1):           tag-name
m.position(1):      10
m[2].str():         the value
m.str(2):           the value
m.position(2):      19
m[3].str():         tag-name
m.str(3):           tag-name
m.position(3):      30

matches:
<tag-name>the value</tag-name> (length: 30)
tag-name (length: 8)
the value (length: 9)
tag-name (length: 8)
```

Иначе говоря, существуют четыре способа вычисления соответствующей строки в целом в объекте `match_result<> m`:

```
m.str()         // вычисляет соответствующую строку в целом
m.str(0)        // то же самое
m[0].str()      // то же самое
*(m.begin())    // то же самое
```

и три способа вычисления *n*-й соответствующей подстроки, если она есть:

```
m.str(1)        // вычисляет первую соответствующую подстроку, если она есть,
                // или "" в противном случае
m[1].str()      // то же самое
*(m.begin()+1) // вычисляет первую соответствующую подстроку, если она есть,
                // в противном случае результат некорректный
```

Если вызвать `regex_match()` вместо `regex_search()`, то интерфейс объекта типа `match_results` будет таким же. Однако, поскольку функция `regex_match()` всегда сравнивает всю символьную строку, префикс и суффикс всегда будут пустыми.

Теперь у нас есть вся информация, необходимая для *всех* соответствий регулярного выражения, как показано в следующей программе:

```
// regex/regex3.cpp

#include <string>
#include <regex>
#include <iostream>
using namespace std;

int main()
{
   string data = "<person>\n"
                 " <first>Nico</first>\n"
                 " <last>Josuttis</last>\n"
                 "</person>\n";

   regex reg("<(.*)>(.*)</(\\1)>");

   // перебор всех соответствий
   auto pos=data.cbegin();
   auto end=data.cend();
   smatch m;
   for ( ; regex_search(pos,end,m,reg); pos=m.suffix().first) {
        cout << "match:  " << m.str() << endl;
        cout << " tag:   " << m.str(1) << endl;
        cout << " value: " << m.str(2) << endl;
   }
}
```

Здесь мы используем регулярное выражение (дополнительная обратная косая черта в строчном литерале C++ игнорируется) `<(.*)>(.*)</(\1)>` для поиска строки

```
<anyNumberOfAnyChars1>anyNumberOfAnyChars2</anyNumberOfAnyChars1>
```

Таким образом, мы ищем дескрипторы XML (`\1` значит: *то же, что и первая соответствующая подстрока*).

В нашем примере данное регулярное выражение используется с помощью другого интерфейса, выполняющего перебор соответствующих символьных строк. По этой причине вместо передачи всей символьной строки мы передаем диапазон соответствующих элементов. Мы начинаем с диапазона, содержащего все символы, используя функции `cbegin()` и `cend()` в строке поиска:

```
auto pos=data.cbegin();
auto end=data.cend();
```

Затем, после каждого сопоставления, мы продолжаем поиск с начала оставшихся символов:

```
smatch m;
for ( ; regex_search(pos,end,m,reg); pos=m.suffix().first) {
   ...
}
```

Таким образом, поскольку анализируемая строка `data` имеет значение

```
<person>
 <first>Nico</first>
 <last>Josuttis</last>
</person>
```

программа выводит следующий результат:

```
match: <first>Nico</first>
tag: first
value: Nico
match: <last>Josuttis</last>
tag: last
value: Josuttis
```

Для повторной инициализации итератора `pos` можно было бы передать аргумент `m[0].second()` (конец сопоставляемых символов), а не выражение `m.suffix().first`. В обоих случаях следует использовать итераторы `const_iterator`. Таким образом, компилятор не допустит использование функции `begin()` и `end()` для инициализации итераторов `pos` и `end`.

Вывод может получиться другим, если дескрипторы в строке `data` не разделены символом перехода на другую строку:

```
<person><first>Nico</first><last>Josuttis</last></person>
```

Итак, вывод может получиться таким:

```
match: <person><first>Nico</first><last>Josuttis</last></person>
tag: person
value: <first>Nico</first><last>Josuttis</last>
```

Причина заключается в том, что функции для работы с регулярными выражениями являются *жадными*. Иначе говоря, возвращается *самое длинное соответствие*. При использовании символов перехода на новую строку дескриптор, открывающийся символами `<person>`, может не совпасть с искомым, потому что мы ищем "`.*`" в качестве значения, т.е. "любое количество любых символом, за исключением перехода на новую строку". Без использования символов перехода на новую строку весь дескриптор с символами `<person>` соответствует шаблону. Чтобы гарантировать, что мы сможем по-прежнему находить внутренние дескрипторы, мы должны были бы изменить регулярное выражение, например:

```
"<(.*)>([^>]*)</(\\1)>"
```

В качестве значения мы теперь ищем группу "[^>]*", т.е. "любое количество любых символов, кроме >". Следовательно, внутренние дескрипторы больше не считаются частью значения.

14.3. Итераторы регулярных выражений

Для того чтобы перебрать все соответствия при регулярном поиске, можно использовать итераторы регулярных выражений. Эти итераторы имеют тип `regex_iterator<>` и обычные конкретизации для строки и символьных последовательностей с префиксами `s`, `c`, `ws` или `wc`. Рассмотрим следующий пример:

```
// regex/regexiter1.cpp

#include <string>
#include <regex>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
   string data = "<person>\n"
                 " <first>Nico</first>\n"
                 " <last>Josuttis</last>\n"
                 "</person>\n";

   regex reg("<(.*)>(.*)</(\\1)>");

   // перебор всех соответствий (с помощью итератора regex_iterator):
   sregex_iterator pos(data.cbegin(),data.cend(),reg);
   sregex_iterator end;
   for ( ; pos!=end ; ++pos ) {
        cout << "match:  " << pos->str() << endl;
        cout << " tag:   " << pos->str(1) << endl;
        cout << " value: " << pos->str(2) << endl;
   }

   // использование итератора regex_iterator для обработки каждой
   // соответствующей подстроки как элемента в алгоритме
   sregex_iterator beg(data.cbegin(),data.cend(),reg);
   for_each (beg,end,[](const smatch& m) {
             cout << "match:  " << m.str() << endl;
             cout << " tag:   " << m.str(1) << endl;
             cout << " value: " << m.str(2) << endl;
   });
}
```

Здесь выражение

```
sregex_iterator pos(data.cbegin(),data.cend(),reg);
```

инициализирует итератор регулярных выражений, выполняющий обход данных для поиска соответствий reg. Конструктор по умолчанию для этого типа определяет итератор, установленный на позицию, следующую за последней:

```
sregex_iterator end;
```

Теперь этот итератор можно использовать как любой другой двунаправленный итератор (см. раздел 9.2.4): операция * возвращает текущее соответствие, а операции ++ и -- перемещают итератор на следующее или предыдущее соответствие. Таким образом, следующие операторы выводят все соответствия, их дескрипторы и значения (как в предыдущем примере):

```
for ( ; pos!=end ; ++pos ) {
     cout << "match:  " << pos->str() << endl;
     cout << " tag:   " << pos->str(1) << endl;
     cout << " value: " << pos->str(2) << endl;
}
```

Разумеется, такой итератор можно использовать и в алгоритме. Таким образом, следующий фрагмент программы выполняет вызов лямбда-функции, передаваемой как третий аргумент, при каждом соответствии (подробное описание лямбда-функций и алгоритмов см. в разделе 6.9):

```
// использование итератора regex_iterator для обработки каждой
// соответствующей подстроки как элемента в алгоритме
sregex_iterator beg(data.cbegin(),data.cend(),reg);
sregex_iterator end;
for_each (beg,end,[](const smatch& m) {
          cout << "match:  " << m.str() << endl;
          cout << " tag:   " << m.str(1) << endl;
          cout << " value: " << m.str(2) << endl;
});
```

14.4. Итераторы токенов регулярных выражений

Итератор регулярных выражений позволяет перебирать все соответствующие подстроки. Однако иногда необходимо обрабатывать все содержимое между соответствующими выражениями. Например, такая ситуация часто возникает, когда требуется разбить строки на отдельные токены, разделенные чем-нибудь, возможно, даже заданным как регулярное выражение. Эту функциональную возможность предоставляет класс regex_token_iterator<>, имеющий обычные конкретизации для строки символьных последовательностей с префиксами s, c, ws или wc.

И вновь, для инициализации мы можем передать начало и конец символьной последовательности и регулярное выражение. Кроме того, можно задать список целочисленных значений, представляющих элементы "токенизации".

- -1 означает, что нас интересуют все подпоследовательности между соответствующими регулярными выражениями (разделители токенов).

- 0 означает, что нас интересуют все соответствующие регулярные выражения (разделители токенов).
- Все другие значения *n* означают, что нас интересует *n*-е соответствующее подвыражение в регулярных выражениях.

Рассмотрим следующий пример:

```
// regex/regextokeniter1.cpp

#include <string>
#include <regex>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
   string data = "<person>\n"
                 " <first>Nico</first>\n"
                 " <last>Josuttis</last>\n"
                 "</person>\n";

   regex reg("<(.*)>(.*)</(\\1)>");

   // перебираем все соответствия (с помощью итератора regex_token_iterator)
   sregex_token_iterator pos(data.cbegin(),data.cend(), // последовательность
                             reg,                       // разделитель токенов
                             {0,2});                    // 0: полное соответствие,
                                                        // 2: вторая подстрока
   sregex_token_iterator end;
   for ( ; pos!=end ; ++pos ) {
        cout << "match: " << pos->str() << endl;
   }
   cout << endl;

   string names = "nico, jim, helmut, paul, tim, john paul, rita";
   regex sep("[ \t\n]*[,;.][ \t\n]*"); // разделенные , ; или . и пропусками
   sregex_token_iterator p(names.cbegin(),names.cend(), // последовательность
                           sep,                         // разделитель
                           -1);        // -1: значения между разделителями
   sregex_token_iterator e;
   for ( ; p!=e ; ++p ) {
        cout << "name: " << *p << endl;
   }
}
```

Программа выводит следующий результат:

```
match: <first>Nico</first>
match: Nico
match: <last>Josuttis</last>
match: Josuttis
```

```
name: nico
name: jim
name: helmut
name: paul
name: tim
name: john paul
name: rita
```

Здесь итератор токенов регулярных выражений для объектов класса `string` (префикс `s`) инициализируется символьной последовательностью *data*, регулярным выражением `reg` и списком из двух индексов (`0` и `2`):

```
sregex_token_iterator pos(data.cbegin(),data.cend(), // последовательность

                          reg,                       // разделитель токенов

                          {0,2});                    // 0: полное соответствие,
                                                     // 2: вторая подстрока
```

Список индексов указывает, что нас интересуют все соответствия и вторые подстроки каждого соответствия.

Обычное применение такого итератора токенов регулярных выражений демонстрирует следующая итерация. В данном случае используется список имен:

```
string names = "nico, jim, helmut, paul, tim, john paul, rita";
```

Теперь регулярное выражение определяет разделители этих имен. В данном случае это запятая, или точка с запятой, или точка с необязательными пропусками (пробелами, символами табуляции и символами перехода на новую строку):

```
regex sep("[ \t\n]*[,;.][ \t\n]*"); // разделенные , ; или . и пропусками
```

В качестве альтернативы можно использовать следующее регулярное выражение (см. раздел 14.8):

```
regex sep("[[:space:]]*[,;.][[:space:]]*"); // разделенные , ; или .
                                            // и пропусками
```

Поскольку нас интересуют только значения между разделителями токенов, программа выводит все имена из списка (удаляя пробелы).

Отметим, что интерфейс класса `regex_token_iterator` позволяет задавать искомые токены разными способами.

- Можно передать отдельное целочисленное значение.
- Можно передать список инициализации или целочисленные значения (см. раздел 3.1.3).
- Можно передать объект класса `vector`, состоящий из целочисленных значений.
- Можно передать массив целочисленных значений.

14.5. Замена регулярных выражений

В заключение рассмотрим интерфейс, позволяющий заменять символьные последовательности, соответствующие регулярному выражению. Рассмотрим следующий пример:

```
// regex/regexreplace1.cpp

#include <string>
#include <regex>
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
   string data = "<person>\n"
                 " <first>Nico</first>\n"
                 " <last>Josuttis</last>\n"
                 "</person>\n";
   regex reg("<(.*)>(.*)</(\\1)>");

   // выводим данные с заменой соответствий шаблонам
   cout << regex_replace (data,                    // данные
                          reg,                     // регулярное выражение
                          "<$1 value=\"$2\"/>")    // замена
        << endl;

   // то же самое с помощью синтаксиса sed
   cout << regex_replace (data,                          // данные
                          reg,                           // регулярное выражение
                          "<\\1 value=\"\\2\"/>",        // замена
                          regex_constants::format_sed)   // флаг формата
        << endl;

   // используем интерфейс итераторов и    // - format_no_copy: не копируем
символы, которые не соответствуют шаблону
   // - format_first_only: заменяем только первое найденное соответствие
   string res2;
   regex_replace (back_inserter(res2),               // получатель
                  data.begin(), data.end(),          // диапазон источника
                  reg,                               // регулярное выражение
                  "<$1 value=\"$2\"/>",              // замена
                  regex_constants::format_no_copy |  // флаги формата
                  regex_constants::format_first_only);
   cout << res2 << endl;
}
```

Здесь мы снова используем регулярное выражение для сравнения значений с XML/HTML-дескрипторами. Однако на этот раз мы преобразовываем ввод в следующий вывод:

```
<person>
 <first value="Nico"/>
 <last value="Josuttis"/>
```

```
</person>

<person>
 <first value="Nico"/>
 <last value="Josuttis"/>
</person>

<first value="Nico"/>
```

Для этого задаем замену, в которой можно использовать соответствующие шаблонам подвыражения с символом `$` (см. табл. 14.1). Здесь `$1` и `$2` используют дескриптор и значение, найденное в заменяющей строке:

```
"<$1 value=\"$2\"/>"  // замена с помощью синтаксиса по умолчанию
```

Неформатированная строка позволяет избежать использования кавычек:

```
R"(<$1 value="$2"/>)" // замена с помощью синтаксиса по умолчанию
```

Передавая константу `regex_constants::format_sed`, можно воспользоваться синтаксисом замены команды UNIX `sed` (см. второй столбец в табл. 14.1).

```
"<\\1 value=\"\\2\"/>" // замена с помощью синтаксиса sed
```

И снова с помощью неформатированной строки мы можем избежать использования обратных косых черт:

```
R"(<\1 value="\2"/>)" // замена с помощью синтаксиса sed,
                      // заданная как неформатированная строка
```

Таблица 14.1. Символы замены в регулярных выражениях

Шаблон по умолчанию		Смысл шаблона sed
`$&`	`&`	Шаблон сравнения
`$`*n*	`\`*n*	*n*-я группа захвата для сравнения
`$'`		Префикс шаблона сравнения
`$'`		Суффикс шаблона сравнения
`$$`		Символ `$`

14.6. Флаги регулярных выражений

Мы уже ввели некоторые константы регулярных выражений, с помощью которых можно влиять на работу интерфейса регулярных выражений.

```
regex reg3("<\\(.*\\)>.*</\\1>",regex_constants::grep); // используем
                                                        // грамматику grep

regex_replace (data, reg,
               string("<\\1 value=\"\\2\"/>"),
               regex_constants::format_sed) // используем синтаксис замены sed
```

Однако этих констант много больше. В табл. 14.2 перечислены все константы регулярных выражений, содержащиеся в библиотеке regex, а также указано, где их можно использовать. В принципе их всегда можно передать конструктору или функции регулярного выражения как необязательный последний аргумент.

Таблица 14.2. Константы регулярных выражений в пространстве имен `std::regex_constants`

Константы regex_constant	Смысл
Грамматика регулярных выражений	
`ECMAScript`	Используется грамматика ECMAScript (по умолчанию)
`Basic`	Используется грамматика базовых регулярных выражений (BRE) системы POSIX
`Extended`	Используется грамматика расширенных регулярных выражений (ERE) системы POSIX
`Awk`	Используется грамматика инструмента UNIX `awk`
`Grep`	Используется грамматика инструмента UNIX `grep`
`Egrep`	Используется грамматика инструмента UNIX `egrep`
Другие флаги создания	
`icase`	Нечувствительность к регистру
`nosubs`	Не хранить подпоследовательности в результатах соответствий
`optimize`	Оптимизировать скорость сравнения, а не скорость создания регулярного выражения
`collate`	Диапазоны символов в виде `[a-b]` будут зависеть от локального контекста
Флаги алгоритма	
`match_not_null`	Пустая последовательность не будет соответствием
`match_not_bol`	Первый символ не будет соответствовать *началу строки* (шаблон `^`)
`match_not_eol`	Последний символ не будет соответствовать *концу строки* (шаблон `$`)
`match_not_bow`	Первый символ не будет соответствовать *началу слова* (шаблон `\b`)
`match_not_eow`	Последний символ не будет соответствовать *концу слова* (шаблон `\b`)
`match_continuous`	Выражение будет соответствовать только подпоследовательности, которая начинается с первого символа
`match_any`	Если есть несколько соответствий, то приемлемым считается любое
`match_prev_avail`	Позиции перед первым символом считаются корректными (игнорируют константы `match_not_bol` и `match_not_bow`)
Флаги замены	
`format_default`	Использовать синтаксис замены по умолчанию (ECMAScript)
`format_sed`	Использовать синтаксис замены инструмента UNIX `sed`
`format_first_only`	Заменять только первое соответствие
`format_no_copy`	Не копировать символы, не соответствующие шаблону

Рассмотрим небольшую программу, демонстрирующую применение некоторых флагов:

```
// regex/regex4.cpp

#include <string>
#include <regex>
#include <iostream>
using namespace std;

int main()
{
   // поиск записей в предметном указателе LaTeX без учета регистра
   string pat1 = R"(\\.*index\{([^}]*)\})";      // первая группа захвата
   string pat2 = R"(\\.*index\{(.*)\}\{(.*)\})"; // 2-я и 3-я группы захвата
   regex pat (pat1+"\n"+pat2,
              regex_constants::egrep|regex_constants::icase);

   // инициализируем строку символами из стандартного ввода
   string data((istreambuf_iterator<char>(cin)),
               istreambuf_iterator<char>());

   // ищем и выводим соответствующие записи в предметном указателе
   smatch m;
   auto pos = data.cbegin();
   auto end = data.cend();
   for ( ; regex_search (pos,end,m,pat); pos=m.suffix().first) {
        cout << "match: " << m.str() << endl;
        cout << " val:  " << m.str(1)+m.str(2) << endl;
        cout << " see:  " << m.str(3) << endl;
   }
}
```

Цель программы — найти записи в предметном указателе LATEX, которые могут иметь один или два аргумента. Кроме того, записи могут быть набраны в нижнем или верхнем регистре. Итак, требуется найти запись, соответствующую следующим условиям.

- Обратная косая черта, за которой следуют какие-то символы и слово `index` (в нижнем или верхнем регистре), а также запись предметного указателя, заключенная в фигурные скобки, как показано ниже.

  ```
  \index{STL}%
  \MAININDEX{standard template library}%
  ```

- Обратная косая черта, за которой следуют какие-то символы и слово `index` (в верхнем или нижнем регистре), а также запись предметного указателя и слова "see also", заключенные в фигурные скобки, как показано ниже.

  ```
  \SEEINDEX{standard template library}{STL}%
  ```

Используя грамматику `egrep`, между двумя регулярными выражениями можно поместить символ перехода на новую строку. (Фактически команды `grep` и `egrep` могут одновременно искать несколько регулярных выражений, заданных в разных строках.) Однако в этом случае следует учитывать свойство *жадности*, т.е. необходимо гарантировать, что первое регулярное выражение не соответствует последовательностям, которые соответствуют второму регулярному выражению. Итак, вместо того, чтобы допускать любые

символы в записи предметного указателя, мы должны гарантировать, что фигурных скобок не будет. В результате получаются следующие регулярные выражения:

```
\\.*index\{([^}]*)\}
\\.*index\{(.*)\}\{(.*)\}
```

Их можно также представить в виде неформатированных строк

```
R"(\\.*index\{([^}]*)\})"
R"(\\.*index\{(.*)\}\{(.*)\})"
```

или обычных строковых литералов

```
"\\\\.*index\\{([^}]*)\\}"
"\\\\.*index\\{(.*)\\}\\{(.*)\\}"
```

Окончательное регулярное выражение мы создаем, выполняя конкатенацию обоих выражений и передавая флаги для использования грамматики, в которой символ `\n` разделяет альтернативные шаблоны (см. раздел 14.9) и игнорируется регистр.

```
regex pat (pat1+"\n"+pat2,
           regex_constants::egrep|regex_constants::icase);
```

В качестве входной информации мы используем все символы, считанные из стандартного потока ввода. Здесь мы используем строку `data`, которая инициализируется началом и концом всех считанных символов (см. разделы 7.1.2 и 15.13.2).

```
string data((istreambuf_iterator<char>(cin)),
             istreambuf_iterator<char>());
```

Отметим, что первое регулярное выражение имеет одну "группу захвата", а второе — две. Таким образом, если первое регулярное выражение соответствует записи предметного указателя, то эта запись оказывается в первой подгруппе. Если записи предметного указателя соответствует второе регулярное выражение, то эта запись окажется во втором частичном соответствии, а слова "see also" — в третьем. По этой причине мы выводим в качестве найденного значения содержимое первого частичного соответствия, а затем второго (одно частичное соответствие будет содержать значение, а второе будет пустым).

```
smatch m;
auto pos = data.begin();
auto end = data.end();
for ( ; regex_search (pos,end,m,pat); pos=m.suffix().first) {
     cout << "match: " << m.str() << endl;
     cout << " val:  " << m.str(1)+m.str(2) << endl;
     cout << " see:  " << m.str(3) << endl;
}
```

Вызовы `str(2)` и `str(3)` являются корректными, даже если соответствий не существует. Функция `str()` в этом случае обязательно выдаст пустую строку.

При вводе строк

```
\chapter{The Standard Template Library}
\index{STL}%
\MAININDEX{standard template library}%
\SEEINDEX{standard template library}{STL}%
This is the basic chapter about the STL.
\section{STL Components}
```

```
\hauptindex{STL, introduction}%
The \stl{} is based on the cooperation of
...
```

программа выводит на экран следующий результат:

```
match: \index{STL}
  val: STL
  see:
match: \MAININDEX{standard template library}
  val: standard template library
  see:
match: \SEEINDEX{standard template library}{STL}
  val: standard template library
  see: STL
match: \hauptindex{STL, introduction}
  val: STL, introduction
  see:
```

14.7. Исключения, связанные с регулярными выражениями

При разборе регулярных выражений ситуация может стать очень сложной. Стандартная библиотека C++ содержит специальный класс исключений для работы с регулярными выражениями. Этот класс является производным от класса `std::runtime_error` (см. раздел 4.3.1) и содержит дополнительную функцию-член `code()` для возвращения кода ошибки. Это может помочь при поиске ошибки в случае возникновения исключения при разборе регулярных выражений.

К сожалению, коды ошибок, возвращаемые функцией `code()`, зависят от реализации, поэтому выводить их на экран не имеет смысла. Вместо этого необходимо использовать следующий заголовочный файл (или подобный ему) для разумной обработки исключений, связанных с регулярными выражениями:

```
// regex/regexexception.hpp

#include <regex>
#include <string>

template <typename T>
std::string regexCode (T code)
{
   switch (code) {
      case std::regex_constants::error_collate:
         return "error_collate: "
                "regex has invalid collating element name";
      case std::regex_constants::error_ctype:
         return "error_ctype: "
                "regex has invalid character class name";
      case std::regex_constants::error_escape:
         return "error_escape: "
                "regex has invalid escaped char. or trailing escape";
      case std::regex_constants::error_backref:
```

```
            return "error_backref: "
                   "regex has invalid back reference";
         case std::regex_constants::error_brack:
            return "error_brack: "
                   "regex has mismatched '[' and ']'";
         case std::regex_constants::error_paren:
            return "error_paren: "
                   "regex has mismatched '(' and ')'";
         case std::regex_constants::error_brace:
            return "error_brace: "
                   "regex has mismatched '{' and '}'";
         case std::regex_constants::error_badbrace:
            return "error_badbrace: "
                   "regex has invalid range in {} expression";
         case std::regex_constants::error_range:
            return "error_range: "
                   "regex has invalid character range, such as '[b-a]'";
         case std::regex_constants::error_space:
            return "error_space: "
                   "insufficient memory to convert regex into finite state";
         case std::regex_constants::error_badrepeat:
            return "error_badrepeat: "
                   "one of *?+{ not preceded by valid regex";
         case std::regex_constants::error_complexity:
            return "error_complexity: "
                   "complexity of match against regex over pre-set level";
         case std::regex_constants::error_stack:
            return "error_stack: "
                   "insufficient memory to determine regex match";
      }
      return "unknown/non-standard regex error code";
   }
```

Подробное объяснение, записанное в скобках после имени кода ошибки, взято непосредственно из спецификации стандартной библиотеки C++. Использование этого заголовочного файла демонстрируется в следующей программе:

```
// regex/regex5.cpp

#include <regex>
#include <iostream>
#include "regexexception.hpp"
using namespace std;

int main()
{
   try {
     // инициализируем регулярное выражение
     // некорректной синтаксической конструкцией
     regex pat ("\\\\.*index\\{([^}]*)\\}",
     regex_constants::grep|regex_constants::icase);
     ...
   }
   catch (const regex_error& e) {
   cerr << "regex_error: \n"
        << " what(): " << e.what() << "\n"
```

```
        << " code(): " << regexCode(e.code()) << endl;
    }
}
```

Поскольку мы используем грамматику `grep`, но прибегаем к управляющим символам перед фигурными скобками, `{` и `}`, программа может вывести следующий результат:

```
regex_error:
what(): regular expression error
code(): error_badbrace: regex has invalid range in {} expression
```

14.8. Грамматика ECMAScript

Грамматикой по умолчанию в библиотеке регулярных выражений является модифицированная грамматика ECMAScript (см. [ECMAScript]) — самая мощная среди всех грамматик. В табл. 14.3 перечислены наиболее важные из специальных выражений и указан их смысл.

Таблица 14.3. Общие регулярные выражения для грамматики по умолчанию (ECMAScript)

Выражение	Смысл
`.`	Любой символ, кроме символа перехода на новую строку
`[...]`	Один из символов ... (может содержать диапазоны)
`[^...]`	Ни один из символов ... (может содержать диапазоны)
`[[:`*charclass*`:]]`	Символ указанного символьного класса *charclass* (см. табл. 14.4)
`\n`, `\t`, `\f`, `\r`, `\v`	Символ перехода на новую строку, табуляции, прогона страницы, перевода каретки, возврата или вертикальной табуляции
`\x`*hh*, `\u`*hhh*	Шестнадцатеричный символ или символ системы Unicode
`\d`, `\D`, `\s`, `\S`, `\w`, `\W`	Сокращение для символа символьного класса (см. табл. 14.4)
`*`	Предыдущий символ или группа произвольное количество раз
`?`	Предыдущий символ или группа — необязательно (ни разу или один раз)
`+`	Предыдущий символ или группа хотя бы один раз
`{n}`	Предыдущий символ или группа *n* раз
`{n,}`	Предыдущий символ или группа хотя бы *n* раз
`{n,m}`	Предыдущий символ или группа не менее *n* и не более *m* раз
`...|...`	Шаблон до символа `|` или шаблон после него
`(...)`	Группировка
`\1`, `\2`, `\3`, ...	*n*-я группа (первая группа имеет индекс 1)
`\b`	Положительная граница слова (начало или конец слова)
`\B`	Отрицательная граница слова (не начало и не конец слова)
`^`	Начало строки (включая начало всех символов)
`$`	Конец строки (включая конец всех символов)

В квадратных скобках можно указывать любую комбинацию символов (включая специальные символы), диапазоны символов (например, `[0-9a-z]`) и символьные классы (например, `[[:digit:]]`). Начальный символ `^` отрицает все выражение, так что все выражение в квадратных скобках означает "любой символ, за исключением ...". В табл. 14.4 приведены все возможные символьные классы для регулярных выражений. Отметим, что базовые классы соответствуют вспомогательным функциям для символьных классификаций в разделе 16.4.4. Однако однобуквенные сокращения поддерживаются только регулярными выражениями. Управляющие последовательности классов символов поддерживаются только грамматикой ECMAScript.

Ниже приведено несколько примеров

```
[_[:alpha:]][_[:alnum:]]*        // идентификатор C++
(.|\n)*                          // любое количество любых символов
                                 // (включая переход на новую строку)
[123]?[0-9]\.1?[0-9]\.20[0-9]{2} // дата первого века второго тысячелетия
                                 // (немецкий формат, например, 24.12.2010)
```

Таблица 14.4. Символьные классы и соответствующие управляющие последовательности (ECMAScript)

Символ	Сокращенное название класса	Управляющая последовательность	Описание
`[[:alnum:]]`			Буква или цифра (эквивалент `[[:alpha:][:digit:]]`)
`[[:alpha:]]`			Буква
`[[:blank:]]`			Пробел или табуляция
`[[:cntrl:]]`			Управляющий символ
`[[:digit:]]`	`[[:d:]]`	`\d`	Цифра
		`\D`	Не цифра (эквивалент `[^[:digit:]]`)
`[[:graph:]]`			Печатаемый непробельный символ (эквивалент `[[:alnum:][:punct:]]`)
`[[:lower:]]`			Буква в нижнем регистре
`[[:print:]]`			Печатаемый символ (включая пропуски)
`[[:punct:]]`			Знаки пунктуации (т.е. печатаемые символы, но не пробел, цифра или буква)
`[[:space:]]`	`[[:s:]]`	`\s`	Пробел
		`\S`	Не пробел (эквивалент `[^[:space:]]`)
		`[[:upper:]]`	Буква в верхнем регистре
`[[:xdigit:]]`			Шестнадцатеричная буква
	`[[:w:]]`	`\w`	Буква, цифра или символ подчеркивания (эквивалент `[[:alpha:][:digit:]_]`)
		`\W`	Не буква, не цифра и не символ подчеркивания (эквивалент `[^[:alpha:][:digit:]_]`)

14.9. Другие грамматики

Помимо грамматики ECMAScript, стандартная библиотека C++ поддерживает пять других грамматик, которые можно задать с помощью соответствующих констант регулярных выражений (см. раздел 14.6).

- `ECMAScript`: грамматика ECMAScript, установленная по умолчанию;
- `basic`: грамматика базовых регулярных выражений (BRE) POSIX;
- `extended`: грамматика расширенных регулярных выражений (ERE) POSIX;
- `awk`: грамматика инструмента UNIX `awk`;
- `grep`: грамматика инструмента UNIX `grep`;
- `egrep`: грамматика инструмента UNIX `egrep`.

Основные различия между этими грамматиками приведены в табл. 14.5. Как видим, грамматика ECMAScript намного мощнее остальных. Немногочисленные возможности, которые она не поддерживает, — это использование символа перехода на новую строку для разделения нескольких шаблонов с помощью операции "или," как это принято в грамматиках `grep` и `egrep`, и возможность инструмента `awk` задавать восьмеричные управляющие последовательности.

Таблица 14.5. Различия между грамматиками регулярных выражений

Возможность	ECMA-Script	basic	extended	awk	grep	egrep
Символы для группировки	`( )`	`\( \)`	`( )`	`( )`	`\( \)`	`( )`
Символы для повторений	`{ }`	`\{ \}`	`{ }`	`{ }`	`\{ \}`	`{ }`
`?` значит "ни разу или один раз"	Да	–	Да	Да	–	Да
`+` значит "хотя бы один раз"	Да	–	Да	Да	–	Да
`|` значит "или"	Да	–	Да	Да	–	Да
`\n` разделяет альтернативные шаблоны	–	–	–	–	Да	Да
`\n` ссылается на группу n	Да	Да	–	–	Да	–
Границы слова (`\b` and `\B`)	Да	–	–	–	–	–
Шестнадцатеричный или Unicode управляющие последовательности	Да	–	–	–	–	–
Управляющие последовательности символьных классов	Да	–	–	–	–	–
`\n`, `\t`, `\f`, `\r`, `\v`	Да	–	–	Да	–	–
`\a` (сигнал) или `\b` (возврат)	–	–	–	Да	–	–
`\ooo` для восьмеричных значений	–	–	–	Да	–	–

14.10. Подробное описание основных сигнатур регулярных выражений

В табл. 14.6 приведены сигнатуры основных операций над регулярными выражениями `regex_match()` (см. раздел 14.1), `regex_search()` (см. раздел 14.1) и `regex_replace()` (см. раздел 14.5). Легко видеть, что существуют перегрузки для операций над строками, которые могут быть как объектами класса `basic_string<>`, так и обычными С-строками, например строковыми литералами, а также для итераторов, задающих начало и конец обрабатываемой последовательности. Кроме того, всегда можно задать формат в качестве необязательного последнего аргумента.

Если соответствие найдено, то функции `regex_match()` и `regex_search()` возвращают значение `true`. Кроме того, они позволяют передавать необязательный аргумент *matchRet*, содержащий детальную информацию о найденных соответствиях. Эти аргументы типа `std::match_results<>` (см. раздел 14.2) должны конкретизироваться типом итератора, соответствующим символьному типу.

- Для строк C++ он соответствует константному итератору. Для типов `string` и `wstring` соответствующие типы `smatch` и `wsmatch` определяются следующим образом:

```
typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;
```

- Для С-строк, включая строковые литералы, он соответствует типу указателя. Для С-строк, состоящих из символов типа `char` или `wchar_t`, определены соответствующие типы `cmatch` и `wcmatch`.

```
typedef match_results<const char*> cmatch;
typedef match_results<const wchar_t*> wcmatch;
```

Для функции `regex_replace()` необходимо передать спецификацию *repl* в виде строки (для нее также существуют перегруженные варианты как для класса `basic_string<>`, так и для обычных С-строк, например строковых литералов). Версия для строки возвращает новую строку с соответствующими заменами. Версия для итератора возвращает первый аргумент *outPos*, который должен быть выходным итератором, указывающим место для записи с замещением.

Таблица 14.6. Сигнатуры операций над регулярными выражениями

Сигнатура	Описание
`bool regex_match(`*str*, *regex*`)`	Проверяет полное соответствие *regex*
`bool regex_match(`*str*, *regex*, *flags*`)`	
`bool regex_match(`*beg*, *end*, *regex*`)`	
`bool regex_match(`*beg*, *end*, *regex*, *flags*`)`	
`bool regex_match(`*str*, *matchRet*, *regex*`)`	

Окончание табл. 14.6

Проверяет и возвращает полное соответствие *regex*	

`bool regex_match(`*str*, *matchRet*, *regex*, *flags*`)`
`bool regex_match(`*beg*, *end*, *matchRet*, *regex*`)`
`bool regex_match(`*beg*, *end*, *matchRet*, *regex*, *flags*`)`
	Проверяет и возвращает полное соответствие *regex*

`bool regex_search(`*str*, *regex*`)`
`bool regex_search(`*str*, *regex*, *flags*`)`
`bool regex_search(`*beg*, *end*, *regex*`)`
`bool regex_search(`*beg*, *end*, *regex*, *flags*`)`
`bool regex_search(`*str*, *matchRet*, *regex*`)`
	Ищет соответствие *regex*

`bool regex_search(`*str*, *matchRet*, *regex*, *flags*`)`
`bool regex_search(`*beg*, *end*, *matchRet*, *regex*`)`
`bool regex_search(`*beg*, *end*, *matchRet*, *regex*, *flags*`)`
strRes `regex_replace(`*str*, *regex*, *repl*`)`
	Ищет и возвращает соответствие *regex*

strRes `regex_replace(`*str*, *regex*, *repl*, *flags*`)` *outPos* `regex_replace(`*outPos*, *beg*, *end*, *regex*, *repl*`)` *outPos* `regex_replace(`*outPos*, *beg*, *end*, *regex*, *repl*, *flags*`)`	Заменяет соответствия согласно *regex*

В заключение заметим, что, для того чтобы избежать недоразумений, стандарт не предусматривает неявное преобразование строк и строковых литералов в тип `regex`. Таким образом, программист всегда должен самостоятельно явно преобразовывать строки, содержащие регулярное выражение, в тип `std::regex` (или `std::basic_regex<>`).

Глава 15

Классы потоков ввода-вывода

Классы для ввода-вывода являются важной частью стандартной библиотеки C++; программы без ввода-вывода не имеют широкого применения. Классы ввода-вывода из стандартной библиотеки C++ не ограничены файлами или экранами и клавиатурой и образуют расширяемую платформу для форматирования произвольных данных и доступа к произвольным внешним представлениям.

Библиотека IOStream, названная так по имени ввода-вывода, — единственная часть стандартной библиотеки C++, которая широко использовалась до стандартизации C++ 98. Ранние дистрибутивы систем C++ поставлялись вместе с рядом классов, разработанных в компании AT&T, которая установила фактический стандарт ввода-вывода. Несмотря на то что эти классы претерпели несколько изменений, чтобы соответствовать стандартной библиотеке C++ и удовлетворять новым потребностям, основные принципы библиотеки IOStream остаются неизменными.

В начале этой главы приведен общий обзор самых важных компонентов и методов, а затем наглядно демонстрируется практическое использование библиотеки IOStream. Ее использование варьируется от простого форматирования до интеграции новых внешних представлений — тема, которой обычно уделяется мало внимания.

Мы не пытаемся обсудить все аспекты библиотеки IOStream подробно; для этого потребовалась бы отдельная книга. За дополнительной информацией обратитесь к одной из книг, посвященных потоковой библиотеке ввода-вывода, или к справочникам по стандартной библиотеке C++.

Выражаю особую благодарность Дитмару Кюлю (Dietmar Kühl), эксперту по вводу-выводу и интернационализации стандартной библиотеки C++, который дал мне много ценных советов и написал первые части этой главы.

Новшества в стандарте C++11

Стандарт C++ 98 определял почти все функции библиотеки IOStream. Ниже представлен список самых важных функциональных возможностей, добавленных в стандарт C++ 11.

- Добавлено несколько новых манипуляторов: `hexfloat` и `defaultfloat` (см. раздел 15.7.6), `get_time()` и `put_time()` (см. раздел 16.4.3), а также `get_money()` и `put_money()` (см. раздел 16.4.2).
- Для того чтобы предоставлять больше информации об исключениях, класс исключений выводится из класса `std::system_error`, а не непосредственно из класса `std::exception` (см. раздел 15.4.4).
- Строковый поток и классы файловых потоков поддерживают rvalue-семантику и семантику перемещения. Таким образом, можно перемещать конструкции, выполнять перемещающее присваивание и менять местами строковый и файловые потоки. Они также обеспечивают возможность использовать временные строковые или файловые потоки для ввода-вывода (см. разделы 15.9.2 и 15.10.2).

- Файловые потоки также позволяют передавать файловые имена в виде объектов `std::string` для файловых имен, а не только как `const char*` (см. раздел 15.9.1).
- Операторы ввода и вывода `<<` и `>>` также перегружены для типов `long long` и `unsigned long long`.
- Потоки ввода-вывода частично поддерживают параллельную работу (см. раздел 15.2.2).
- Свойства символов распространены на типы `char16_t` и `char32_t` (см. раздел 16.1.4).
- С помощью нового класса `wbuffer_convert` можно настроить потоки на чтение и запись разных наборов символов, например UTF-8 (см. раздел 16.4.4).

15.1. Основы потоков ввода-вывода

Прежде чем перейти к подробному описанию потоковых классов, кратко обсудим общие аспекты потоков, заложив основу для дальнейшей работы. Читатели, знакомые с потоковым вводом-выводом, могут пропустить этот раздел.

15.1.1. Потоковые объекты

В языке C++ ввод-вывод выполняется при помощи потоков данных. *Поток* — это *поток данных*, содержащий символы, которые "*текут*". Согласно принципам объектно-ориентированного программирования, *поток* — это объект со свойствами, определенными классом. *Вывод* интерпретируется как запись данных в поток, а *ввод* — как чтение данных из потока. Для стандартных каналов ввода-вывода существуют стандартные глобальные объекты.

15.1.2. Классы потоков

В соответствии с разными видами ввода-вывода (ввод, вывод и доступ к файлам) существуют разные классы, зависящие от вида ввода-вывода. Перечислим главные классы потоков.

- **Класс `istream`** определяет потоки ввода, которые можно использовать для чтения данных.
- **Класс `ostream`** определяет потоки вывода, которые можно использовать для вывода данных.

Оба класса представляют собой специализации шаблонных классов `basic_istream<>` и `basic_ostream<>` соответственно с помощью символьного типа `char`. Библиотека IOStream не зависит от конкретного символьного типа. Вместо этого в большинстве классов библиотеки IOStream символьный тип используется как шаблонный аргумент. Эта параметризация является аналогом параметризации строковых классов и используется для интернационализации (подробно эта тема будет рассмотрена в главе 16).

В этом разделе мы рассмотрим ввод и вывод в узких потоках, потоках данных типа `char`. Потоки, работающие с другими символьными типами, мы обсудим позднее в данной главе.

15.1.3. Глобальные потоковые объекты

В библиотеке IOStream определено несколько глобальных объектов типов `istream` и `ostream`, которые соответствуют стандартным каналам ввода-вывода.

- **Объект `cin`** класса `istream` — стандартный канал ввода, используемый для ввода пользовательских данных. Этот поток соответствует потоку `stdin` в языке C. Обычно операционная система связывает этот поток с клавиатурой.
- **Объект `cout`** класса `ostream` — стандартный поток вывода, используемый для вывода результатов работы программы. Этот поток соответствует потоку `stdout` в языке C. Обычно операционная система связывает этот поток с монитором.
- **Объект `cerr`** класса `ostream`— стандартный канал ошибок, используемый для всех видов сообщений об ошибках. Этот поток соответствует потоку `stderr` в языке C. Обычно операционная система связывает его с монитором. По умолчанию поток `cerr` не буферизуется.
- Объект **`clog`** класса `ostream` — стандартный поток регистрации. У него нет эквивалентов в языке C. По умолчанию этот поток связывается с тем же получателем, что и поток `cerr`, но при этом данные в потоке `clog` буферизуются.

Отделение обычного вывода от вывода сообщений об ошибках позволяет по-разному выводить эти данные при выполнении программ. Например, обычный вывод программы можно перенаправить в файл, а сообщения об ошибках — на консоль. Разумеется, для этого операционная система должна поддерживать перенаправление стандартных каналов ввода-вывода (большинство операционных систем поддерживают эту возможность). Разделение стандартных каналов происходит от концепции перенаправления ввода-вывода системы UNIX.

15.1.4. Потоковые операторы

Операторы сдвига >> и << перегружены для потоковых классов, поэтому они называются операторами ввода и вывода[1]. При помощи этих операторов ввод-вывод можно выполнять "цепочкой".

Например, следующий цикл при каждой итерации считывает два целых числа из стандартного потока ввода, поскольку пока мы ограничимся вводом только целых чисел, и записывает их в стандартный поток вывода:

```
int a, b;

// пока ввод чисел a и b выполняется успешно
while (std::cin >> a >> b) {
    // вывод чисел a и b
    std::cout << "a: " << a << " b: " << b << std::endl;
}
```

15.1.5. Манипуляторы

В конце большинства операторов вывода записывается так называемый манипулятор:

```
std::cout << std::endl
```

[1] Поскольку эти операторы вставляют символы в поток или извлекают символы из потока, иногда их называют операциями *вставки в поток вывода* и *извлечения из потока ввода*.

Манипуляторы — это специальные объекты, предназначенные для управления потоком данных. Обычно манипуляторы изменяют только интерпретацию ввода или форматирование вывода (например, манипуляторы выбора системы счисления `dec`, `hex` и `oct`). Таким образом, манипуляторы потока данных `ostream` не всегда осуществляют вывод, а манипуляторы потока данных `istream` не всегда потребляют вводимые данные. Однако некоторые манипуляторы служат триггерами непосредственных действий, например, очистки буфера вывода или переключения в режим игнорирования пропусков в буфере ввода.

Манипулятор `endl` обозначает "конец строки" и выполняет две операции.

1. Выводит символ перехода на новую строку (т.е. символ `\n`).
2. Выполняет очистку буфера вывода (т.е. принудительно выводит все буферизованные данные из потока с помощью метода `flush()`).

Наиболее важные манипуляторы библиотеки IOStream приведены в табл. 15.1. Более подробно манипуляторы, в том числе манипуляторы, определенные в библиотеке IOStream, рассматриваются в разделе 15.6. Там же показано, как создавать свои манипуляторы.

Таблица 15.1. Наиболее важные манипуляторы из библиотеки IOStream

Манипулятор	Класс	Описание
`endl`	`ostream`	Выводит символ `'\n'` и очищает буфер вывода
`ends`	`ostream`	Выводит `'\0'`
`flush`	`ostream`	Очищает буфер вывода
`ws`	`istream`	Считывает и игнорирует пропуски

15.1.6. Простой пример

Использование потоковых классов иллюстрируется следующим примером. Эта программа считывает два числа с плавающей точкой и выводит их произведение:

```
// io/io1.cpp

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

int main()
{
  double x, y; // операнды

  // выводим строку заголовка
   cout << "Multiplication of two floating point values" << endl;

  // выводим первый операнд
   cout << "first operand: ";
   if (! (cin >> x)) {
      // ошибка ввода
      // => сообщение об ошибке и выход из программы с возвращением кода ошибки
      cerr << "error while reading the first floating value"
           << endl;
```

```
    return EXIT_FAILURE;
  }

  // читаем второй операнд
  cout << "second operand: ";
  if (! (cin >> y)) {
    // ошибка ввода
    // => сообщение об ошибке и выход из программы с возвращением кода ошибки
    cerr << "error while reading the second floating value"
         << endl;
    return EXIT_FAILURE;
  }

  // выводим операнды и результат
  cout << x << " times " << y << " equals " << x * y << endl;
}
```

15.2. Основные потоковые классы и объекты

15.2.1. Иерархия классов

Потоковые классы образуют иерархию, показанную на рис. 15.1. Для шаблонных классов в верхней строке показано его имя, а в нижней строке — имена специализации для символьных типов `char` и `wchar_t`.

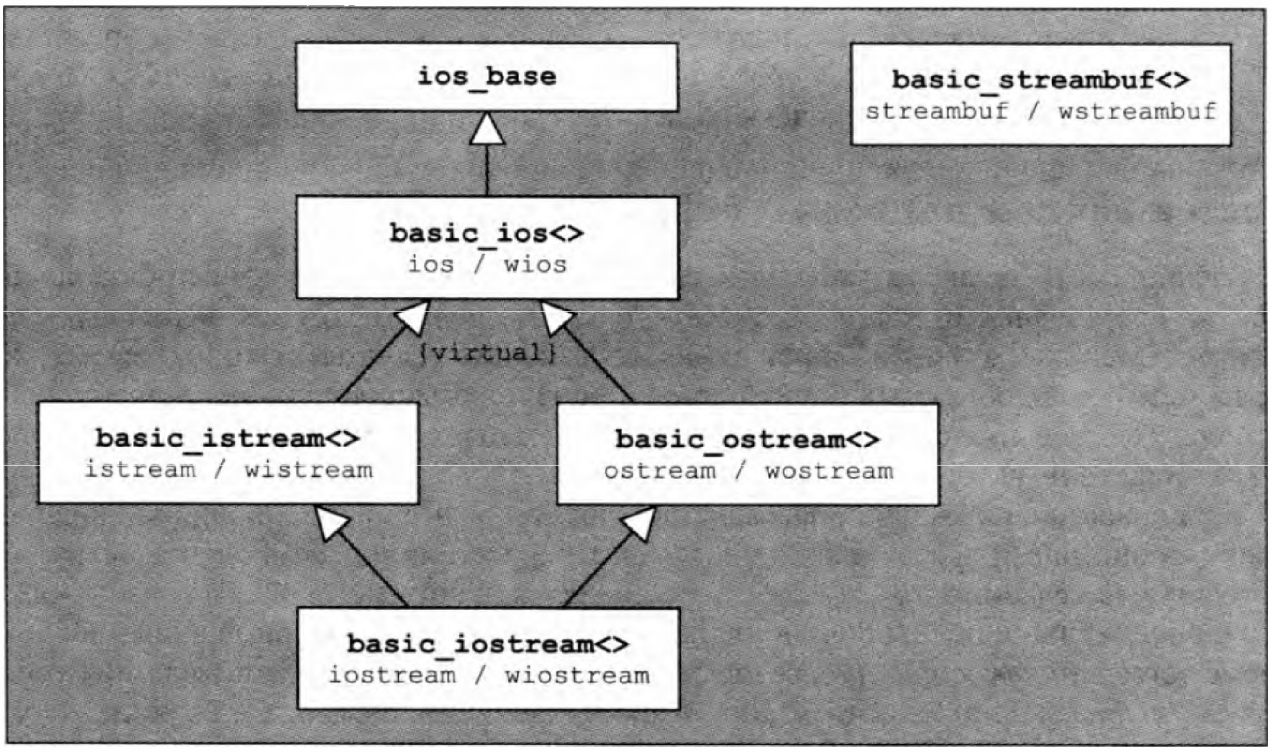


Рис. 15.1. Иерархия основных потоковых классов

Классы в этой иерархии выполняют следующие задачи.

- Базовый класс **ios_base** определяет свойства всех потоковых классов, не зависящие от типа и соответствующих свойств символов. Этот класс в основном состоит из компонентов и функций, предназначенных для управления состоянием и флагами формата.
- Шаблонный класс **basic_ios<>**, производный от класса ios_base, определяет общие свойства всех потоковых классов, зависящие от типа и соответствующих свойств символов. К этим свойствам относится также определение буфера, используемого потоком данных. Буфер — это объект класса, производного от базового класса basic_streambuf<>, с соответствующей специализацией. Именно он выполняет операции чтения и/или записи.
- Шаблонные классы **basic_istream<>** и **basic_ostream<>**, виртуально производные от класса basic_ios<>, определяют объекты, которые могут использоваться для чтения и записи соответственно. Эти классы, как и классы basic_ios<>, представляют собой шаблоны, параметризованные символьным типом и его свойствами. Если проблемы интернационализации не имеют значения, используются специализации этих классов для символьного типа char — классы istream и ostream.
- Шаблонный класс **basic_iostream<>** является производным от двух шаблонных классов — basic_istream<> и basic_ostream<>. Он определяет объекты, которые могут использоваться как для чтения, так и для записи.
- Шаблонный класс **basic_streambuf<>** является ядром библиотеки IOStream. Он определяет интерфейс всех представлений, которые могут быть записаны в потоки или считаны из потоков, и используется другими потоковыми классами для чтения или записи символов. Для получения доступа к некоторым внешним представлениям классы объявляются производными от класса basic_streambuf<>. Подробности приведены в следующих разделах.

Предназначение потоковых буферных классов

Библиотека IOStream разрабатывалась на основе строгого разделения обязанностей. Классы, производные от класс basic_ios, предназначены только для *форматирования* данных[2]. Операции чтения и записи символов выполняются потоковыми буферами, поддерживаемыми подобъектами класса basic_ios. Потоковые буфера обеспечивают чтение и запись в символьных буферах. Кроме того, они позволяют абстрагироваться от внешнего представления, например, файлов или строк.

Таким образом, потоковые буфера играют важную роль при выполнении ввода-вывода с новыми внешними представлениями (например, сокетами или компонентами графического пользовательского интерфейса), перенаправлении потоков данных или их объединении в конвейер (например, при сжатии данных вывода перед их записью в другой поток). Кроме того, потоковые буфера обеспечивают синхронизацию ввода-вывода при работе с одним внешним представлением. Детально этот метод рассматривается в разделе 15.12.

[2] На самом деле они не выполняют даже форматирования! Форматирование делегируется соответствующим фацетам в библиотеке локальных контекстов. Подробно фацеты описаны в разделах 16.2.2 и 16.4.

Потоковые буфера упрощают определение новых внешних представлений, например нового хранилища данных. Для этого достаточно объявить новый потоковый буферный класс, производный от класса `basic_streambuf<>` (или его подходящей специализации), и определить функции чтения и/или записи символов для нового внешнего представления. Все возможности форматированного ввода-вывода автоматически становятся доступными, когда объект потока данных инициализируется для использования объекта нового потокового буферного класса. Потоковые буфера подробно описываются в разделе 15.13, а примеры определения новых потоковых буферов для доступа к специальным устройствам хранения приведены в разделе 15.13.3.

Подробные определения классов

Как и все шаблонные классы в библиотеке IOStream, шаблонный класс `basic_ios<>` параметризован двумя аргументами и определен следующим образом:

```
namespace std {
   template <typename charT,
             typename traits = char_traits<charT> >
            class basic_ios;
}
```

Шаблонными аргументами являются символьный тип, используемый потоковыми классами, и класс, описывающий свойства этого символьного типа.

Примерами свойств, определенными в классе свойств, являются значения, используемые как признак файла[3], и инструкции о том, как копировать или перемещать последовательности символов. Как правило, свойства символьного типа ассоциируются с определенным символьным типом, поэтому целесообразно определить шаблонный класс, специализируемый для конкретных символьных типов. Следовательно, класс свойств по умолчанию является классом `char_traits<charT>`, если аргумент задает символьный тип `charT`. Стандартная библиотека C++ содержит специализации класса `char_traits` для символьных типов `char`, `char16_t`, `char32_t` и `wchar_t`.[4] Более подробно свойства символов описаны в разделе 16.1.4.

Существуют две специализации класса `basic_ios<>` для двух наиболее распространенных символьных типов:

```
namespace std {
   typedef basic_ios<char> ios;
   typedef basic_ios<wchar_t> wios;
}
```

Тип `ios` соответствует базовому классу старой библиотеки IOStream, разработанной компанией AT&T, и может использоваться для обеспечения совместимости со старыми программами на языке C++.

Класс потокового буфера, используемый классом `basic_ios`, определяется аналогично.

---

[3] Термин *конец файла* я использую как синоним термина *конец входных данных*. Он соответствует константе `EOF` в языке C.

[4] Свойства символов для типов `char16_t` и `char32_t` введены стандартном C++11.

```
namespace std {
   template <typename charT,
             typename traits = char_traits<charT> >
            class basic_streambuf;

   typedef basic_streambuf<char> streambuf;
             typedef basic_streambuf<wchar_t> wstreambuf;
}
```

Разумеется, шаблонные классы `basic_istream<>`, `basic_ostream<>` и `basic_iostream<>` также параметризуются символьным типом и классом свойств.

```
namespace std {
   template <typename charT,
             typename traits = char_traits<charT> >
            class basic_istream;

template <typename charT,
          typename traits = char_traits<charT> >
         class basic_ostream;

template <typename charT,
          typename traits = char_traits<charT> >
         class basic_iostream;
}
```

Аналогично другим классам, существуют специализации двух наиболее важных символьных типов:

```
namespace std {
   typedef basic_istream<char> istream;
   typedef basic_istream<wchar_t> wistream;

   typedef basic_ostream<char> ostream;
   typedef basic_ostream<wchar_t> wostream;

   typedef basic_iostream<char> iostream;
   typedef basic_iostream<wchar_t> wiostream;
}
```

Типы `istream` и `ostream` обычно используются в Западном полушарии, где достаточно использовать наборы восьмибитовых символов[5]. Тип `wchar_t` позволяет использовать наборы символов, состоящих из более чем восьми битов (см. раздел 16.1). Для типов `char16_t` и `char32_t` в стандартной библиотеке C++ нет соответствующих специализаций.

Стандартная библиотека C++ содержит дополнительные потоковые классы для форматированного ввода-вывода в файлы (см. раздел 15.9) и строки (см. раздел 15.10).

---

[5] Классы `istream_withassign`, `ostream_withassign` и `iostream_withassign` (производные от классов `istream`, `ostream` и `iostream`), которые можно найти в некоторых старых потоковых библиотеках, не поддерживаются стандартом. Соответствующие функциональные возможности можно обеспечить по-другому (см. раздел 15.12.3).

15.2.2. Глобальные потоковые объекты

В потоковых классах определено несколько глобальных потоковых объектов, предназначенных для обеспечения доступа к стандартным каналам ввода-вывода с символьными типами `char` и `wchar_t` (табл. 15.2).

Таблица 15.2. Глобальные потоковые объекты

Тип	Имя	Назначение
`istream`	`cin`	Считывает данные из стандартного канала ввода
`ostream`	`cout`	Записывает обычные данные в стандартный канал вывода
`ostream`	`cerr`	Записывает сообщения об ошибках в стандартный канал ошибок
`ostream`	`clog`	Записывает регистрационные сообщения в стандартный канал регистрации
`wistream`	`wcin`	Считывает символы в расширенной кодировке из стандартного канала ввода
`wostream`	`wcout`	Записывает обычные символы в расширенной кодировке в стандартный канал вывода
`wostream`	`wcerr`	Записывает сообщение об ошибках, состоящее из символов в расширенной кодировке, в стандартный канал ошибок
`wostream`	`wclog`	Записывает сообщение регистрации, состоящее из символов в расширенной кодировке, в стандартный канал регистрации

По умолчанию эти стандартные потоки синхронизированы со стандартными потоками в языке С. Иначе говоря, стандартная библиотека C++ гарантирует сохранение порядка смешанного вывода в потоки C++ и С. Прежде чем записать в них данные, каждый буфер стандартных потоков C++ очищает соответствующий буфер С, и наоборот. Разумеется, эта синхронизация требует определенных затрат времени. Если она не нужна, то ее можно отключить с помощью вызова `sync_with_stdio(false)` перед вводом или выводом (см. раздел 15.14.1).

После принятия стандарта C++11 на потоковые объекты были распространены гарантии, касающиеся параллельной работы: при синхронизации со стандартными потоками С использующие их параллельные потоки больше не вызывают непредсказуемых последствий. Таким образом, существует возможность записывать и читать несколько потоков. Тем не менее это может привести к появлению чередующихся символов или неопределенности при чтении символов из потока. Для любых других потоковых объектов или объектов, не синхронизирующихся с потоками С, параллельное чтение или запись приводит к непредсказуемым последствиям.

15.2.3. Заголовочные файлы

Определения потоковых классов разбросаны по нескольким заголовочным файлам.

- **Файл `<iosfwd>`** содержит опережающие объявления потоковых классов. Этот заголовочный файл необходим, потому что простые опережающие объявления вроде `class ostream` теперь запрещены.

- **Файл <streambuf>** содержит определения базового класса потокового буфера (basic_streambuf<>).
- **Файл <istream>** содержит определения классов, поддерживающих только ввод (basic_istream<>) или и ввод, и вывод (basic_iostream<>).[6]
- **Файл <ostream>** содержит определения для потокового класса вывода (basic_ostream<>).
- **Файл <iostream>** содержит объявления глобальных потоковых объектов, таких как cin и cout.

Большинство заголовочных файлов предназначено для внутренней организации стандартной библиотеки C++. Прикладному программисту достаточно включить файл <iosfwd> в объявление потоковых классов и файл <istream> или <ostream> при непосредственном использовании функций ввода или вывода соответственно. Заголовочный файл <iostream> следует включать только при использовании стандартных потоковых объектов. В некоторых реализациях при выполнении каждого модуля, включающего этот заголовочный файл, происходит инициализация. Этот код не связан с большими затратами, но при этом приходится загружать соответствующие страницы исполняемого файла, а эта операция может быть затратной. Как правило, в программу следует включать только совершенно необходимые заголовочные файлы. В частности, заголовочные файлы должны включать только заголовок <iosfwd>, а соответствующие файлы реализации — в заголовок с полным определением.

Для специальных функциональных возможностей, предназначенных для работы с потоками данных (параметризованных манипуляторов, а также файловых и строковых потоков), предназначены дополнительные заголовочные файлы (<iomanip>, <fstream>, <sstream> и <strstream>). Дополнительная информация об этих заголовках приводится в разделах, посвященных этим специальным средствам.

15.3. Стандартные потоковые операторы << и >>

В языках C и C++ операторы << и >> используются для сдвига битов целых чисел вправо и влево соответственно. Классы basic_istream и basic_ostream перегружают операторы >> и << для выполнения стандартного ввода-вывода. Таким образом, в языке C++ операторы сдвига становятся операторами ввода-вывода[7].

15.3.1. Оператор вывода <<

Класс basic_ostream, а также классы ostream и wostream определяют оператор << как оператор вывода и перегружают его практически для всех фундаментальных типов, исключая void и nullptr_t, а также для типов char* и void*.

---

[6] На первый взгляд кажется нелогичным объявлять классы для ввода *и* вывода в файле <istream>. Но так при трансляции каждого модуля, содержащего заголовочный файл <iostream>, происходит инициализация, занимающая определенное время (см. объяснения в следующем абзаце). Объявления для ввода *и* вывода были помещены в файле <istream>.

[7] Иногда операторы ввода-вывода называют операторами *вставки* и *извлечения*.

Операторы вывода для потоков определены так, чтобы соответствующий поток задавался их вторым аргументом. Таким образом, данные посылаются в направлении стрелки:

```
int i = 7;
std::cout << i; // вывод: 7

float f = 4.5;
std::cout << f; // вывод: 4.5
```

Оператор << можно перегрузить так, чтобы второй аргумент имел произвольный тип, позволяя интегрировать пользовательские типы данных с системой ввода-вывода. Компилятор гарантирует, что для вывода второго аргумента будет вызвана правильная функция. Разумеется, эта функция должна преобразовать второй аргумент в последовательность символов, посылаемых в поток.

Стандартная библиотека C++ также использует этот механизм для определения операторов вывода для специальных типов, таких как строки (см. раздел 13.3.10), битовые множества (см. раздел 12.5.1) и комплексные числа (см. раздел 17.2.3).

```
std::string s("hello");
s += ", world";
std::cout << s;                          // вывод: hello, world

std::bitset<10> flags(7);
std::cout << flags;                      // вывод: 0000000111

std::complex<float> c(3.1,7.4);
std::cout << c;                          // вывод: (3.1,7.4)
```

Подробное описание процесса создания операторов вывода для пользовательских типов данных приведено в разделе 15.11.

Тот факт, что механизм вывода можно расширить на пользовательские типы данных, является важным усовершенствованием по сравнению с механизмом ввода-вывода в языке C, использующим функцию `printf()`: нет необходимости указывать тип объекта, который должен выводиться на печать. Вместо этого перегрузка разных типов гарантирует, что для вывода будет автоматически выведена правильная функция. Этот механизм не ограничивается стандартными типами. Таким образом, пользователь имеет только один механизм, который работает со всеми типами.

Оператор << может также использоваться для вывода нескольких объектов в одном операторе. По общепринятому соглашению, операторы вывода возвращают свой первый аргумент. Иначе говоря, результатом оператора вывода является поток вывода. Это позволяет образовывать цепочки операторов вывода:

```
std::cout << x << " times " << y << " is " << x * y << std::endl;
```

Оператор << выполняется слева направо. Таким образом, оператор

```
std::cout << x
```

выполняется первым. Это правило не распространяется на порядок вычисления аргументов — оно определяет только порядок выполнения операторов.

Это выражение возвращает свой первый операнд `std::cout`. Следовательно, следующим выполняется оператор

```
std::cout << " times "
```

Объект y, строковый литерал " is " и результат операции x * y выводятся в соответствующем порядке. Оператор умножения имеет более высокий приоритет, чем оператор <<, поэтому скобки вокруг выражения x * y не нужны. Однако существуют операторы с более низким приоритетом, например логические. В этом примере, если переменные x и y являются числами с десятичной точкой и значениями 2.4 и 5.1, на экран будет выведена строка

```
2.4 times 5.1 is 12.24
```

Отметим, что после принятия стандарта C++11 возможен параллельный вывод, использующий один и тот же потоковый объект, но при этом в результате в выводе могут появляться чередующиеся символы (см. раздел 15.2.2).

15.3.2. Оператор ввода >>

Класс basic_istream, а также классы istream и wistream определяют оператор ввода >>. Аналогично классу basic_ostream, этот оператор перегружен почти для всех фундаментальных типов, за исключением void и nullptr_t, а также для типов char* и void*. Операторы для потоков определены так, чтобы хранить считанное значение во втором аргументе. Как и в операторе <<, данные посылаются в направлении стрелки.

```
int i;
std::cin >> i;  // считываем int из стандартного потока ввода и храним его в i

float f;
std::cin >> f; // считываем float из стандартного потока ввода и храним его в f
```

Отметим, что второй аргумент изменяется. Для того чтобы это стало возможным, второй аргумент передается по неконстантной ссылке.

Как и в случае оператора <<, существует возможность перегрузить оператор ввода для произвольного типа данных и образовывать цепочки операторов:

```
float f;
std::complex<double> c;
std::cin >> f >> c;
```

Для этого ведущие пробельные символы по умолчанию игнорируются. Однако это автоматическое игнорирование можно отключить (подробнее об этом — в разделе 15.7.7).

После принятия стандарта C++11 появилась возможность выполнять параллельный ввод из одного и того же объекта, но при этом иногда невозможно определить, какой поток выполнения какие символы считывает (см. раздел 15.2.2).

15.3.3. Ввод и вывод специальных типов

Стандартные операторы ввода-вывода позволяют работать с любыми фундаментальными типами (за исключением void и nullptr_t), а также с типами char* и void*. Однако для некоторых из этих типов, а также для пользовательских типов установлены специальные правила.

Числовые типы

При чтении числовых типов ввод должен начинаться хотя бы с одной цифры. В противном случае числовое значение будет положено равным `0` и будет установлен флаг `failbit` (см. раздел 15.4.1).

```
int x;
std::cin >> x; // присваиваем 0 переменной x, если следующий
               // символ не соответствует числу
```

Однако, если входных данных нет или флаг `failbit` уже установлен, выполнение оператора ввода не изменяет переменную `x`. Это же относится и к типу `bool`.

Тип `bool`

По умолчанию булевы величины выводятся и считываются как числа: значение `false` преобразовывается в `0` или из `0`, а значение `true` преобразовывается в `1` или из `1`. При чтении значения, отличающиеся от `0` и `1`, считаются ошибочными. В этом случае устанавливается флаг `ios::failbit`, что может привести к генерации соответствующего исключения (см. раздел 15.4.4).

Для потока можно задать режим форматирования, при котором булевы величины вводятся и выводятся как символьные строки (см. раздел 15.7.2). В этом случае возникает проблема интернационализации: если не используются специальные контексты локализации, то значениями булевых величин являются строки `"true"` и `"false"`. В других контекстах могут использоваться другие строки. Например, при использовании немецкого контекста локализации соответствующими строками будут `"wahr"` и `"falsch"`. Подробности изложены в разделе 16.2.2.

Типы `char` и `wchar_t`

При чтении переменных типа `char` или `wchar_t` с помощью оператора >> ведущие пробельные символы по умолчанию игнорируются. Для того чтобы считать все символы, включая пробельные, необходимо либо сбросить флаг `skipws` (см. раздел 15.7.7), либо использовать функцию-член `get()` (см. раздел 15.5.1).

Тип `char*`

С-строка (переменная типа `char*`) вводится по словам. Иначе говоря, при чтении С-строки ведущие пробельные символы по умолчанию игнорируются, а чтение продолжается, пока не будет обнаружен очередной пробельный символ или конец файла. Автоматическим игнорированием ведущих пробельных символов можно управлять с помощью флага `skipws` (см. раздел 15.7.7).

Такое поведение означает, что считываемая строка может быть сколь угодно длинной. Обычно программисты на языке С ошибочно полагают, что строка может состоять из не более 80 символов. Такого ограничения не существует. Следовательно, если строка слишком длинная, программист должен сам управлять досрочным прекращением ее ввода. Для этого *всегда* следует устанавливать максимальную длину считываемых строк. Обычно это выглядит примерно так:

```
char buffer[81];                          // 80 символов и '\0'
std::cin >> std::setw(81) >> buffer;
```

Манипулятор `setw()` и соответствующий потоковый параметр подробно описаны в разделе 15.7.3.

При необходимости разместить длинную строку в объекте типа `string` из стандартной библиотеки C++ (см. главу 13) последний автоматически увеличивается. В отличие от типа `char*`, этот способ намного проще и надежнее. Кроме того, класс `string` предоставляет удобную функцию-член `getline()` для чтения строка за строкой (см. раздел 13.2.10). Итак, по возможности следует избегать C-строк и использовать объекты класса `string`.

Тип `void*`

Операторы << и >> позволяют выводить указатель и вновь считывать его. Если оператор вывода получает аргумент типа `void*`, то он выводит адрес, формат которого зависит от реализации. Например, следующий оператор выводит содержимое C-строки и ее адрес:

```
char* cstring = "hello";
std::cout << "string \"" << cstring << "\" is located at address: "
          << static_cast<void*>(cstring) << std::endl;
```

Результат этого оператора может выглядеть следующим образом:

```
string "hello" is located at address: 0x10000018
```

Можно даже снова прочитать адрес с помощью оператора ввода. Однако адреса обычно являются временными. Один и тот же объект при разных запусках программы может иметь разные адреса. Ввод и вывод адресов может использоваться в программах, обменивающихся адресами для идентификации объектов или использующих общую память.

Потоковые буфера

Операторы << и >> могут выполнять прямое чтение данных из потокового буфера и прямую запись данных в потоковый буфер соответственно. Вероятно, это самый быстрый способ копирования файлов с помощью потоков ввода-вывода в языке C++. Соответствующие примеры приведены в разделе 15.14.3.

Пользовательские типы

В принципе механизм ввода-вывода легко расширяется для пользовательских типов. Однако учесть все доступные варианты форматирования данных и возможные ошибки не так просто, как может показаться. Подробно расширение стандартного механизма ввода-вывода для пользовательских типов будет обсуждаться в разделе 15.11.

Денежные и временные значения

После принятия стандарта C++11 появилась возможность использования манипуляторов для прямого чтения и записи денежных и временных значений.

Например, следующая программа позволяет записывать текущую дату и время и считывать новую дату:

```
// io/timemanipulator1.cpp

#include <iostream>
#include <iomanip>
#include <chrono>
#include <cstdlib>


#include <ctime>
using namespace std;

int main ()
{
   // обрабатываем и выводим на печать текущую дату и время:
   auto now = chrono::system_clock::now();
   time_t t = chrono::system_clock::to_time_t(now);
   tm* nowTM = localtime(&t);
   cout << put_time(nowTM,"date: %x\ntime: %X\n") << endl;

   // считываем дату:
   tm date;
   cout << "new date: ";
   cin >> get_time(&date,"%x");           // считываем дату
   if (!cin) {
       cerr << "invalid format read" << endl;
   }
}
```

Перед тем как запросить новую дату, программа может вывести на экран следующий результат:

```
date: 09/14/11
time: 11:08:52
```

Соответствующие манипуляторы позволяют учитывать международные аспекты представления денежных и временных значений. Подробное описание временных манипуляторов изложены в разделе 16.4.3; денежные манипуляторы описаны в разделе 16.4.2, а временная библиотека, в которой определено системное время `std::chrono::system_time`, — в разделе 5.7.

15.4. Состояние потоков

Потоки поддерживают состояние, которое указывает, был ли ввод или вывод успешным, и если нет — устанавливает причину сбоя.

15.4.1. Константы состояния потоков

Общее состояние потоков определяется несколькими константами типа `iostate`, представляющими собой флаги (табл. 15.3). Тип `iostate` является членом класса `ios_base`.

Точный тип констант зависит от реализации, иначе говоря, стандарт не определяет, является ли тип `iostate` перечислением, целым типом или специализацией класса `bitset`.

Таблица 15.3. Константые типы `iostate`

Константа	Описание
`goodbit`	Все в порядке; остальные биты сброшены
`eofbit`	Обнаружен конец файла
`failbit`	Ошибка; сбой при выполнении операции ввода-вывода
`badbit`	Фатальная ошибка; неопределенное состояние

По определению бит `goodbit` равен `0`. Таким образом, установка флага `goodbit` означает, что все остальные биты сброшены. Имя `goodbit` может в определенной степени вводить в заблуждение, потому что оно на самом деле означает, что все остальные биты сброшены.

Разница между флагами `failbit` и `badbit` состоит в том, что флаг `badbit` служит индикатором более серьезной ошибки.

- **Бит `failbit`** устанавливается, если операция не была выполнена правильно, но поток остался в исправном состоянии. Обычно этот флаг является результатом ошибок форматирования при чтении данных. Например, этот флаг устанавливается, если должно считываться целое число, а следующим символом является буква.
- **Бит `badbit`** устанавливается при повреждении потока или потере данных. Например, этот флаг устанавливается, когда указатель в файловом потоке ссылается на позицию, предшествующую началу файла.

Бит `eofbit` обычно устанавливается одновременно с битом `failbit`, потому что конец файла проверяется и обнаруживается при попытке чтения за концом файла. После считывания последнего символа флаг `eofbit` еще не установлен. Следующая попытка считать символ установит биты `eofbit` *и* `failbit`, потому что чтение невозможно.

Некоторые ранние реализации поддерживали флаг `hardfail`. В настоящее время стандарт не поддерживает этот флаг.

Эти константы определяются не глобально, а локально — в классе `ios_base`. Следовательно, перед их именами необходимо указывать оператор области видимости или имя объекта, например:

```
std::ios_base::eofbit
```

Разумеется, можно также использовать класс, производный от класса `ios_base`. В старых реализациях эти константы были определены в классе `ios`. Поскольку класс `ios` является производным от класса `ios_base` и его имя содержит меньше букв, программисты часто используют такие выражения:

```
std::ios::eofbit
```

Эти флаги поддерживаются классом `basic_ios`, а значит, они существуют во всех объектах типа `basic_istream` или `basic_ostream`. Однако потоковые буфера не имеют флагов состояния. Один потоковый буфер может совместно использоваться несколькими потоковыми объектами, поэтому флаги означают состояние потока во время последней

операции и только при условии, что перед выполнением этой операции был установлен бит goodbit. В противном случае флаги могли бы быть установлены одной из предыдущих операций.

15.4.2. Функции-члены для доступа к состоянию потоков

Текущее состояние флагов можно определить с помощью функций-членов, приведенных в табл. 15.4.

Таблица 15.4. Функции-члены для работы с состоянием потоков

Функция-член	Описание
good()	Возвращает true, если поток находится в работоспособном состоянии (установлен бит goodbit)
eof()	Возвращает true, если обнаружен конец файла (установлен бит eofbit)
fail()	Возвращает true, если обнаружена ошибка (установлен бит failbit или badbit)
bad()	Возвращает true, если обнаружена фатальная ошибка (установлен бит badbit)
rdstate()	Возвращает флаги, установленные в данный момент
clear()	Сбрасывает все флаги
clear(*state*)	Сбрасывает и устанавливает флаги состояния *state*
setstate(*state*)	Устанавливает дополнительные флаги состояния *state*

Первые четыре функции-члена, указанные в табл. 15.4, выясняют определенные состояния и возвращают булево значение. Функция fail() возвращает признак того, что установлен бит failbit или badbit. Несмотря на то что все это делается в основном по историческим причинам, преимущество этих функций заключается в том, что всего лишь одна проверка позволяет определить, возникла ли ошибка.

Кроме того, состояние флагов можно определить и изменить с помощью более общих функций-членов. Когда функция clear() вызывается без параметров, все флаги ошибок, включая eofbit, сбрасываются (именно это означает слово *clear*):

```
// сбрасываем все флаги ошибок (включая eofbit):
strm.clear();
```

Если функция clear() получает параметр, то состояние потока уточняется по этому параметру. Другими словами, для потока устанавливается набор флагов, заданных параметром, а остальные флаги сбрасываются.

Единственным исключением является флаг badbit, который всегда устанавливается, если буфера потока не существует, что происходит, если rdbuf() == 0 (см. раздел 15.12.2).

В следующем примере выполняется проверка флага failbit и его сбрасывание (по необходимости):

```
// проверяем, установлен ли флаг failbit
   if (strm.rdstate() & std::ios::failbit) {
       std::cout << "failbit was set" << std::endl;
```

```
// сбрасываем только флаг failbit
   strm.clear (strm.rdstate() & ~std::ios::failbit);
}
```

В этом примере используются битовые операторы `&` и `~`. Оператор `~` возвращает битовое дополнение своего аргумента. Таким образом, следующее выражение возвращает временное значение, в котором установлены все биты, кроме `failbit`:

```
~ios::failbit
```

Оператор `&` возвращает результат побитового оператора "и", примененного к его операндам. При этом установленными остаются только биты, которые были установлены в обоих операндах. Применение побитового "и" ко всем установленным в данный момент флагам (`rdstate()`) и ко всем флагам, за исключением `failbit`, сохраняет значения всех битов и сбрасывает флаг `failbit`.

Потоки можно конфигурировать так, чтобы они генерировали исключения при установке определенных флагов с помощью функций `clear()` или `setstate()` (см. раздел 15.4.4). Такие потоки всегда генерируют исключение, если соответствующий флаг установлен в конце метода, используемого для манипулирования флагами.

Все биты всегда следует сбрасывать явным образом. В языке C было возможным читать символы после обнаружения ошибки форматирования. Например, если функция `scanf()` не могла прочитать целое число, можно было прочитать остальные символы. Таким образом, операция чтения выполнялась неправильно, но поток ввода оставался в корректном состоянии. В языке C++ ситуация иная. Если установлен бит `failbit`, то каждая следующая потоковая операция становится фиктивной, пока флаг `failbit` не будет сброшен явным образом.

В принципе установленные флаги отражают лишь то, что произошло в прошлом. Если после выполнения операции флаг оказался установленным, это не всегда означает, что причиной стала именно эта операция. Флаг мог быть установлен еще до выполнения этой операции. Следовательно, если неизвестно, установлен ли бит ошибки, то перед выполнением операции следует вызвать функцию `clear()`. Это позволит проанализировать состояние флагов и обнаружить ошибку. Однако после сброса флагов результат выполнения операции может оказаться иным. Например, даже если в результате выполнения операции был установлен флаг `eofbit`, это еще не означает, что после сброса флага `eofbit` при повторном выполнении операции он будет установлен снова. Это объясняется тем, например, что между двумя выполнениями операции файл мог увеличиться.

15.4.3. Состояние потока и булевы условия

Для использования потоков в логических выражениях используются две функции (табл. 15.5).

Таблица 15.5. Операторы для работы с потоками в логических выражениях

Функция-член	Описание
`operator bool ()`	Проверяет, находится ли поток в нормальном состоянии (соответствует `!fail()`)
`operator ! ()`	Проверяет, находится ли поток в ошибочном состоянии (соответствует `fail()`)

Функция `operator bool()`[8] позволяет лаконично проверять текущее состояние потоков в управляющих конструкциях.

```
// проверяем, находится стандартный поток ввода в нормальном состоянии
while (std::cin) {
   ...
}
```

Для проверки логического условия в управляющей конструкции не обязательно непосредственно преобразовывать тип в `bool`. Достаточно одного преобразования в целочисленный тип, например `int` или `char`, или в тип указателя. Преобразование в тип `bool` часто используется для чтения объектов и проверки его успешности в одном и том же выражении.

```
if (std::cin >> x) {
   // чтение x было успешным
   ...
}
```

Как мы уже говорили, следующее выражение возвращает объект `cin`:

```
std::cin >> x
```

Таким образом, после того как будет прочитан объект `x`, проверка выглядит как

```
if (std::cin) {
   ...
}
```

Поскольку объект `cin` используется в контексте проверки условия, вызывается функция `operator void*`, проверяющая, находится ли поток в ошибочном состоянии.

Типичное применение этого способа — цикл, в котором выполняются чтение и обработка объектов:

```
// пока возможно чтение объекта obj
while (std::cin >> obj) {
   // обрабатываем объект obj (в данном случае просто выводим его на печать)
   std::cout << obj << std::endl;
}
```

Это классический фильтр из языка C, примененный к объектам языка C++. Цикл заканчивается, если устанавливается флаг `failbit` или `badbit`. Это происходит, если возникает ошибка или обнаруживается конец файла (попытка прочитать данные за концом файла приводит к установке флагов `eofbit` *и* `failbit`; см. раздел 15.4.1). По умолчанию оператор >> игнорирует ведущие пробельные символы. Именно этого обычно ожидают от программы. Однако, если объект `obj` имеет тип `char`, ведущие пробельные символы обычно считаются значимыми. В этом случае можно использовать функции-члены потоков `put()` или `get()` (см. раздел 15.5.3) или, что еще лучше, итератор `istreambuf_iterator` (см. раздел 15.13.2) для реализации фильтра ввода-вывода.

[8] До принятия стандарта C++11 эта операция объявлялась как `operator void*()`, что могло вызывать проблемы, такие, как, например, описанные в разделе 15.10.1.

Оператор ! позволяет выполнить обратную проверку. Он проверяет, находится ли поток в ошибочном состоянии, т.е. оператор возвращает значение `true`, если установлен флаг `failbit` или `badbit`. Оператор можно использовать следующим образом:

```
if (! std::cin) {
   // поток cin не находится в нормальном состоянии
   ...
}
```

Аналогично неявным преобразованиям булевого значения, этот оператор часто используется для проверки успеха одновременно с чтением объекта в одном выражении.

```
if (! (std::cin >> x)) {
   // сбой при чтении
   ...
}
```

Здесь следующее выражение возвращает объект `cin`, к которому применяется оператор !:

```
std::cin >> x
```

Выражение после оператора ! должно быть заключено в круглые скобки из-за правил приоритета операторов: без круглых скобок оператор ! выполнялся бы первым. Иначе говоря, выражение

```
!std::cin >> x
```

эквивалентно выражению

```
(!std::cin) >> x
```

Вероятно, это не то, к чему стремился программист.

Несмотря на то что эти операторы очень удобно использовать в булевых выражениях, можно заметить одну странность: двойное отрицание *не* возвращает исходный объект.

- `cin` является объектом потока класса `istream`.
- `!!cin` — это булево значение, описывающее состояние потока `cin`.

Как и о многих других функциональных возможностях языка C++, можно спорить, является ли хорошим стилем использование булевых значений. Использование функций-членов, таких как `fail()`, обычно повышает читабельность программ.

```
std::cin >> x;
if (std::cin.fail()) {
    ...
}
```

15.4.4. Состояние потока и исключения

Обработка исключений была включена в язык C++ для реагирования на ошибки и особые ситуации (см. раздел 4.3). Однако это было сделано после того, как потоки данных получили широкое распространение. Для того чтобы обеспечить обратную совместимость, потоки данных по умолчанию не генерируют исключений. В то же время для каждого

флага состояния стандартизированных потоков можно указать, должна ли установка этого флага генерировать исключение. Для этой цели используется функция-член `exceptions()` (табл. 15.6).

Таблица 15.6. Функции-члены для работы с исключениями

Функция-член	Описание
`exceptions(`*flags*`)`	Устанавливает флаги, установка которых генерирует исключение
`exceptions()`	Возвращает флаги, установка которых генерирует исключение

Вызов функции-члена `exceptions()` без аргументов возвращает текущие флаги, установка которых сопровождается генерацией исключений. Если функция возвращает флаг `goodbit`, исключения не генерируются. Этот режим установлен по умолчанию, чтобы обеспечить обратную совместимость. Если функция `exceptions()` вызывается с аргументом, исключение генерируется, если был установлен соответствующий флаг. Если флаг состояния, представляющий собой аргумент функции `exceptions()`, был установлен еще до ее вызова, генерируется исключение.

В следующем примере демонстрируется конфигурирование потока так, чтобы при установке всех флагов генерировались исключения.

```
// генерируем исключения для всех "ошибок"
strm.exceptions (std::ios::eofbit | std::ios::failbit |
                 std::ios::badbit);
```

Если в качестве аргумента был передан `0` или флаг `goodbit`, никакие исключения не генерируются:

```
// исключения не генерируются
strm.exceptions (std::ios::goodbit);
```

Исключения генерируются, когда соответствующие флаги состояния были установлены после вызова функции `clear()` или `setstate()`. Исключение генерируется, даже если флаг уже был установлен и не сброшен:

```
// этот вызов генерирует исключение, если передается флаг failbit
strm.exceptions (std::ios::failbit);
...
// генерируется исключение (даже если флаг failbit уже был установлен)
strm.setstate (std::ios::failbit);
```

Генерируемые исключения представляют собой объекты класса `std::ios_base::failure`. После принятия стандарта C++11 этот класс наследуется от класса `std::system_error` (см. раздел 4.3.1)[9].

```
namespace std {
   class ios_base::failure : public system_error {
   public:
      explicit failure (const string& msg,
                        const error_code& ec = io_errc::stream);
```

[9] До принятия стандарта C++11 класс `std::ios_base::failure` непосредственно выводился из класса `std::exception`.

```
        explicit failure (const char* msg,
                          const error_code& ec = io_errc::stream);
    };
}
```

Реализации должны предусматривать создание объекта `error_code`, содержащего конкретную причину сбоя. Ошибка операционной системы должна иметь категорию `"system"`, возвращаемую функцией `category()`, и код, возвращаемый функцией `value()`. Ошибка, возникшая в библиотеке потоков ввода-вывода, должна иметь категорию `"iostream"`, возвращаемую функцией `category()`, и код `std::io_errc::stream`, возвращаемый функцией `value()`. Подробное описание класса `error_code` и работы с ним см. в разделе 4.3.2.

Отказ от генерации исключений по умолчанию означает, что обработка исключений предназначена для действительно неожиданных ситуаций. Эта обработка относится к *исключениям*, а не к *ошибкам*. Ожидаемые ошибки, например, ошибки форматирования при вводе данных пользователем, считаются "нормальными" и обычно намного лучше обрабатываются с помощью флагов состояний.

Основная область применения потоковых исключений — чтение форматированных данных, например автоматически записанных файлов. Но даже в этих ситуациях обработка исключений вызывает проблемы. Например, если требуется, чтобы данные считывались до конца файла, невозможно получить исключение для ошибок, не получив исключение для конца файла. Причина заключается в том, что обнаружение конца файла приводит к установке флага `failbit`, т.е. чтение объекта не было успешным. Для того чтобы отличить обнаружение конца файла от ошибки ввода, следует проверять состояние потока.

Следующий пример показывает, как это должно выглядеть. Он содержит функцию, считывающую из потока числа с плавающей точкой, пока не будет обнаружен конец файла, и возвращающую сумму считанных чисел:

```
// io/sum1a.cpp

#include <istream>
namespace MyLib {
    double readAndProcessSum (std::istream& strm)
{
        using std::ios;
        double value, sum;

        // сохраняем текущее состояние флагов исключения
        ios::iostate oldExceptions = strm.exceptions();

        // позволяем флагам failbit и badbit генерировать исключения
        // - ПРИМЕЧАНИЕ: failbit также устанавливается при обнаружении конца файла
        strm.exceptions (ios::failbit | ios::badbit);
        try {
            // пока поток в нормальном состоянии
            // - считываем значение и добавляем к переменной sum
            sum = 0;
            while (strm >> value) {
                sum += value;
            }
    }
```

```
    catch (...) {
      // если исключение не вызвано обнаружением конца файла
      // - восстанавливаем старое состояние флагов исключения
      // - повторно генерируем исключение
      if (!strm.eof()) {
          strm.exceptions(oldExceptions); // восстанавливаем флаги исключения
          throw;                          // повторно генерируем исключение
      }
   }

      // восстанавливаем старое состояние флагов исключения
      strm.exceptions (oldExceptions);

      // возвращаем sum
      return sum;
   }
}
```

Сначала функция сохраняет исключения, относящиеся к состоянию потока в объекте `oldExceptions`, чтобы восстановить их позднее. Затем поток конфигурируется так, чтобы генерировать исключение при определенных условиях. В цикле все значения считываются и добавляются к переменной типа `long`, пока поток находится в нормальном состоянии. При обнаружении конца файла поток выходит из нормального состояния и генерируется соответствующее исключение для установки флага `eofbit`. Это происходит потому, что при попытке прочитать несуществующие данные был обнаружен конец файла, что также приводит к установке флага `failbit`. Для того чтобы обнаружение конца файла не сопровождалось генерацией исключения, исключение перехватывается локально с целью проверки состояния потока с помощью функции `eof()`. Исключение передается дальше, только если функция `eof()` возвращает `false`.

Отметим, что восстановление исходных флагов исключений само может генерировать исключение: функция `exceptions()` генерирует исключение, если в потоке уже установлен соответствующий флаг. Таким образом, если состояние сгенерировало исключения для флагов `eofbit`, `failbit` или `badbit` для ввода, эти исключения возвращаются вызывающей стороне.

В простейшем случае эту функцию можно вызывать из главного модуля.

```
// io/summain.cpp

#include <iostream>
#include <exception>
#include <cstdlib>

namespace MyLib {
   double readAndProcessSum (std::istream&);
}

int main()
{
   using namespace std;
   double sum;

   try {
        sum = MyLib::readAndProcessSum(cin);
```

```
    }
    catch (const ios::failure& error) {
       cerr << "I/O exception: " << error.what() << endl;
       return EXIT_FAILURE;
    }
    catch (const exception& error) {
       cerr << "standard exception: " << error.what() << endl;
       return EXIT_FAILURE;
    }
    catch (...) {
       cerr << "unknown exception" << endl;
       return EXIT_FAILURE;
    }

    // выводим sum
    cout << "sum: " << sum << endl;
}
```

Оправданны ли все эти усилия? Ведь с потоками можно работать и не генерируя исключений. В данном случае исключение генерируется, если была обнаружена ошибка. Это создает дополнительное преимущество, позволяя создавать пользовательские сообщения об ошибках и использовать классы ошибок.

```
// io/sum2a.cpp

#include <istream>

namespace MyLib {
   double readAndProcessSum (std::istream& strm)
{
   double value, sum;
   // пока поток в нормальном состоянии
   // - считываем значение и добавляем к переменной sum
   sum = 0;
   while (strm >> value) {
          sum += value;
   }
   if (!strm.eof()) {
       throw std::ios::failure
             ("input error in readAndProcessSum()");
   }

      // возвращаем sum
      return sum;
   }
}
```

Программа стала проще, не так ли?

Исключения ввода-вывода до стандарта C++11

До принятия стандарта C++11 класс `std::ios_base::failure` непосредственно наследовал класс `std::exception` и имел только один конструктор, получающий аргумент класса `std::string`.

```
namespace std {
   class ios_base::failure : public exception {
      public:
        explicit failure (const string& msg);
             ...
      };
}
```

Это создавало следующие ограничения.

- Для объекта генерируемого исключения можно было вызывать функцию `what()` только для того, чтобы получить строку, содержащую информацию об ошибке и зависящую от реализации. Ни категории. ни коды ошибок не поддерживались.
- Поскольку конструктор получал только объект `std::string`, при передаче строкового литерала следовало включать заголовок `<string>`. (Для того чтобы стало возможным преобразование в тип `std::string`, необходимо объявить соответствующий строковый конструктор.)

15.5. Стандартные функции ввода–вывода

Вместо стандартных потоковых операторов (>> и <<) для чтения данных из потока и записи данных в поток можно использовать функции-члены, описанные в этом разделе. Эти функции читают и записывают неформатированные данные, в отличие от операторов << и >>, которые читают и записывают форматированные данные. Функции, описанные в этом разделе, никогда не игнорируют ведущие пробельные символы при чтении. Этим они отличаются от оператора >>, который по умолчанию игнорирует ведущие пробельные символы. Это задание возложено на объект `sentry` (см. раздел 15.5.4). Кроме того, эти функции по-другому обрабатывают исключения: если генерируется исключение, флаг `badbit` устанавливается независимо от того, что стало причиной исключения — вызванная функция или установка флага состояния (см. раздел 15.4.4). Если в маске исключений установлен флаг `badbit`, исключение генерируется повторно.

Для работы со счетчиками стандартные функции ввода-вывода используют тип `streamsize`, определенный в заголовке `<ios>`.

```
namespace std {
   typedef ... streamsize;
   ...
}
```

Тип `streamsize` обычно представляет собой версию типа `size_t` со знаком. Он имеет знак, потому что используется и для задания отрицательных чисел.

15.5.1. Функции-члены для ввода

В следующих определениях слово *istream* обозначает потоковый класс, используемый для чтения. Это может быть класс `istream`, `wistream` или другая специализация шаблонного класса `basic_istream`. Слово *char* обозначает соответствующий символьный тип (`char` для `istream` и `wchar_t` для `wistream`). Другие типы и значения, выводимые

курсивом, зависят от точного определения символьного типа или класса свойств, связанного с потоком.

Стандартная библиотека C++ содержит несколько функций-членов для чтения последовательностей символов. Возможности этих функций сравниваются в табл. 15.7 (буква *s* означает последовательность считываемых символов).

Таблица 15.7. Возможности операций для чтения символов из потока

Функция-член	Признак конца чтения	Количество символов	Добавление завершающего символа	Тип возвращаемого значения
`get(`*s*`,`*num*`)`	Исключая новую строку или конец файла	До `num-1`	Да	`istream`
`get(`*s*`,`*num*`,`*t*`)`	Исключая *t* или конец файла	До `num-1`	Да	`istream`
`getline(`*s*`,`*num*`)`	Включая новую строку или конец файла	До `num-1`	Да	`istream`
`getline(`*s*`,`*num*`,`*t*`)`	Включая *t* или конец файла	До `num-1`	Да	`istream`
`read(`*s*`,`*num*`)`	Конец файла	`num`	Нет	`istream`
`readsome(`*s*`,`*num*`)`	Конец файла	До `num`	Нет	Количество символов

int **`istream::get ()`**

- Считывает следующий символ.
- Возвращает считанный символ или *EOF*.
- В общем случае типом возвращаемого значения является `traits::int_type`, а *EOF* — это значение, возвращаемое функцией `traits::eof()`. Для класса `istream` типом возвращаемого значения является `int`, а *EOF* — это константа `EOF`. Таким образом, для класса `istream` эта функция соответствует функции `getchar()` или `getc()` из языка C.
- Возвращаемое значение не обязательно имеет символьный тип, но может иметь тип с более широким диапазоном значений. В противном случае было бы невозможно отличить *EOF* от символов с соответствующим значением.

***istream*& `istream::get (`*char*& *c*`)`**

- Присваивает следующий символ аргументу *c*.
- Возвращает поток. Состояние потока означает, было ли чтение символа успешным.

***istream*& `istream::get (`*char* str*`, streamsize` *count*`)`**
***istream*& `istream::get (`*char* str*`, streamsize` *count*`,` *char delim*`)`**

- Обе формы считывают до *count*`-1` символов в последовательность символов, на которую ссылается указатель *str*.

- Первая форма прекращает чтение, если следующий символ, подлежащий считыванию, является символом перехода на новую строку в соответствующем наборе символов. Для класса `istream` этим символом является `'\n'`, а для класса `wistream` им является `wchar_t('\n')` (см. раздел 16.1.5). Обычно используется символ `widen('\n')` (см. раздел 15.8).
- Вторая форма прекращает чтение, если очередным считанным символом является *delim*.
- Обе формы возвращают поток. Состояние потока означает, было ли чтение символов успешным.
- Символ прекращения чтения (*delim*) не считывается.
- К считанной последовательности символов добавляется нулевой (завершающий) символ.
- Вызывающая сторона должна гарантировать, что строка *str* является достаточно большой, чтобы вместить *count* символов.

istream& **istream::getline** (*char* str*, streamsize *count*)
istream& **istream::getline** (*char* str*, streamsize *count*, *char delim*)

- Обе формы идентичны предыдущим аналогам функции `get()`, за исключением следующего.
  - Они прекращают чтение, *включая* символ перехода на новую строку или *delim* соответственно, а не до их обнаружения. Таким образом, символ перехода на новую строку или *delim* считываются, если они обнаруживаются среди *count*-1 символов, но *не сохраняются* в строке *str*.
  - Если функции считывают строку, состоящую из более чем *count*-1 символов, то они устанавливают флаг `failbit`.

istream& **istream::read** (*char* str*, streamsize *count*)

- Считывает *count* символов в строку *str*.
- Возвращает поток. Состояние потока означает, было ли чтение успешным.
- К строке *str не* добавляется автоматически (завершающий) нулевой символ.
- Вызывающая сторона должна гарантировать, что строка *str* является достаточно большой, чтобы вместить *count* символов.
- Обнаружение конца файла во время чтения считается ошибкой и сопровождается установкой флага `failbit` в дополнение к флагу `eofbit`.

streamsize **istream::readsome** (*char* str*, streamsize *count*)

- Считывает до *count* символов в строку *str*.
- Возвращает количество считанных символов.
- К строке *str не* добавляется автоматически (завершающий) нулевой символ.
- Вызывающая сторона должна гарантировать, что строка *str* является достаточно большой, чтобы вместить *count* символов.

- В отличие от функции `read()`, функция `readsome()` считывает все доступные символы из потокового буфера, используя функцию-член буфера `in_avail()` (см. раздел 15.13.1). Это полезно, когда ожидание ввода нежелательно, потому что он поступает от клавиатуры или от других процессов. Обнаружение конца файла не считается ошибкой и не сопровождается установкой ни флага `eofbit`, ни флага `failbit`.

`streamsize` ***`istream`*`::gcount`** `() const`

- Возвращает количество символов, считанных последней операций *неформатированного* чтения.

***`istream`*`& istream::ignore`** `()`
***`istream`*`& istream::ignore`** `(streamsize` *`count`*`)`
***`istream`*`& istream::ignore`** `(streamsize` *`count`*`, int` *`delim`*`)`

- Обе формы извлекают и игнорируют символы.
- Первая форма игнорирует один символ.
- Вторая форма игнорирует до *count* символов.
- Третья форма игнорирует до *count* символов, пока не будет извлечен и проигнорирован *delim*.
- Если *count* равен `std::numeric_limits<std::streamsize>::max()` (максимальному значение типа `std::streamsize`; см. раздел 5.3), то все символы игнорируются, пока не будет обнаружен *delim* или признак конца файла.
- Все формы возвращают поток.
- Примеры:
  - следующий вызов игнорирует оставшуюся часть строки:

```
cin.ignore(numeric_limits<std::streamsize>::max(),'\n');
```

  - следующий вызов игнорирует весь остаток объекта `cin`:

```
cin.ignore(numeric_limits<std::streamsize>::max());
```

***`int istream`*`::peek`** `()`

- Возвращает следующий символ, подлежащий считыванию из потока, без его извлечения. Этот символ будет прочитан при следующем считывании (если позиция считывания не была изменена).
- Возвращает *EOF*, если больше нельзя считать ни один символ.
- *EOF* — это значение, возвращаемое функцией `traits::eof()`. Для класса `istream` оно является константой `EOF`.

***`istream`*`& istream::unget`** `()`
***`istream`*`& istream::putback`** `(`*`char c`*`)`

- Обе функции возвращают последний считанный символ обратно в поток, чтобы считать его при следующем чтении (если позиция чтения не была изменена).

- Разница между функциями `unget()` и `putback()` заключается в том, что функция `putback()` выполняет проверку, является ли аргумент *c* последним считанным символом.
- Если символ не может быть возвращен в поток или функция `putback()` возвращает в поток неправильный символ, устанавливается флаг `badbit`, который может генерировать соответствующее исключение (см. раздел 15.4.4).
- Максимальное количество символов, которые можно вернуть в поток с помощью этих функций, не регламентировано. Стандарт гарантирует правильность работы только одного вызова этих функций между двумя операциями чтения.

При чтении С-строк безопаснее использовать функции из этого раздела, чем оператор >>. Причина заключается в том, что максимальная длина строки, которую можно прочитать, должна явно передаваться в виде аргумента. Хотя существует возможность ограничивать количество символов, считываемых с помощью оператора >> (см. раздел 15.7.3), об этом легко забыть.

Вместо использования функций-членов потоков часто удобнее работать с потоковым буфером непосредственно. Потоковые буфера имеют функции-члены для эффективного чтения отдельных символов или последовательностей символов без дополнительных затрат на создание объектов `sentry` (см. раздел 15.5.4). Интерфейс потоковых буферов подробно описывается в разделе 15.13. Кроме того, можно использовать шаблонный класс `istreambuf_iterator<>`, предоставляющий интерфейс итератора к потоковому буферу (см. раздел 15.13.2).

Существуют еще две функции для манипулирования позицией ввода — `tellg()` и `seekg()`, которые скорее относятся к работе с файлами (они описаны в разделе 15.9.4).

15.5.2. Функции-члены для вывода

В следующих определениях слово *ostream* обозначает потоковый класс, используемый для записи. Это может быть класс `ostream`, `wostream` или другая специализация класса шаблона `basic_ostream<>`. Слово *char* обозначает соответствующий символьный тип (`char` для `ostream` и `wchar_t` для `wostream`). Другие типы или значения, выделенные курсивом, зависят от точного определения символьного типа или класса свойств, связанного с потоком.

***ostream*& *ostream*::put** (*char c*)

- Записывает аргумент *c* в поток.
- Возвращает поток. Состояние потока определяет, была ли запись успешной.

***ostream*& *ostream*::write** (const *char* str*, streamsize *count*)

- Записывает *count* символов строки *str* в поток.
- Возвращает поток. Состояние потока означает, была ли запись успешной.
- (Заключительный) нулевой символ *не* рассматривается как завершающий строку и может выводиться в поток.
- Вызывающая сторона должна гарантировать, что строка *str* содержит по крайней мере *count* символов, в противном случае поведение неопределено.

***ostream*& *ostream*::flush ()**

- Очищает буферы потока, выполняя запись всех буферизованных данных на устройство или в канал ввода-вывода, которому принадлежит функция.

Существуют еще две функции, модифицирующие позицию записи: `tellp()` и `seekp()`. Они относятся в основном к работе с файлами и будут описаны в разделе 15.9.4.

Как и при работе с функциями ввода, для неформатированной записи может быть целесообразно непосредственно использовать буфер потока (см. раздел 15.14.3) или использовать шаблонный класс `ostreambuf_iterator<>` (см. раздел 15.13.2). В принципе функции для неформатированной записи не дают особых преимуществ, за исключением того, что они используют объекты `sentry` (см. раздел 15.5.4), которые, например, используются для синхронизации потоков вывода (см. раздел 15.12.1).

15.5.3. Примеры использования

Напишем программу на языке C++, реализующую классическую схему фильтрации данных в среде C/UNIX, которая просто записывает все прочитанные символы.

```
// io/charcat1.cpp

#include <iostream>
using namespace std;

int main()
{
   char c;

   // пока можно читать символы
   while (cin.get(c)) {
      // выводим символ
      cout.put(c);
   }
}
```

При каждом выполнении выражения, указанного ниже, следующий символ присваивается переменной `c`, передаваемой по ссылке.

```
cin.get(c)
```

Функция `get()` возвращает поток, таким образом, оператор `while` проверяет, находится ли поток `cin` в нормальном состоянии[10].

Для обеспечения более высокой эффективности можно работать прямо с буферами потоков. Соответствующая версия программы, в которой для ввода и вывода используются итераторы потоковых буферов, приведена в разделе 15.13.2, а версия, копирующая весь ввод с помощью одного оператора, — в разделе 15.14.3.

[10] Этот интерфейс лучше обычного интерфейса языка C для фильтров. На языке C следует использовать функции `getchar()` или `getc()`, возвращающие следующий символ или признак конца файла. Это вызывает проблемы, потому что возвращаемое значение должно обрабатываться как целочисленное, чтобы отличать значения типа `char` от признака конца файла.

15.5.4. Объекты класса sentry

Операторы и функции потокового ввода-вывода используют общую схему: сначала выполняется предварительная обработка потока для ввода-вывода, затем выполняется реальный ввод-вывод, после чего осуществляется заключительная обработка.

Для реализации этой схемы в классах `basic_istream` и `basic_ostream` определен вспомогательный класс `sentry`. Конструкторы этих классов выполняют предварительную обработку, а деструкторы — соответствующую заключительную обработку[1]. Таким образом, все операторы форматированного и неформатированного ввода-вывода и соответствующие функции используют объект `sentry` до реальной обработки данных.

```
sentry se(strm); // неявная предварительная и заключительная обработка
if (se) {
   ... // реальная обработка
}
```

Конструктор объекта `sentry` получает аргумент в виде потока `strm`, который должен подвергнуться предварительной и заключительной обработке. Остальная обработка зависит от состояния этого объекта, служащего индикатором того, что поток находится в нормальном состоянии. Это состояние можно проверить, преобразовав объект `sentry` в тип `bool`. Для потоков ввода объект `sentry` можно создать с необязательным дополнительным булевым значением, служащим индикатором того, следует ли отменить игнорирование ведущих пробельных символов, даже если установлен флаг `skipws`.

```
sentry se(strm,true); // не игнорировать ведущие пробельные
                      // символы при дополнительной обработке
```

Предварительная и заключительная обработка выполняют все общие задачи для потоков ввода-вывода. К этим задачам относятся синхронизация нескольких потоков, проверка того, находится ли поток в нормальном состоянии, игнорирование ведущих пробельных символов, а также возможные специфические задачи, зависящие от реализации. Например, в многопоточной среде может использоваться дополнительная обработка для блокировки.

Если оператор ввода-вывода непосредственно манипулирует буфером потока, сначала необходимо создать объект `sentry`.

15.6. Манипуляторы

Манипуляторы для потоков, упомянутые в разделе 15.1.5, представляют собой объекты, которые модифицируют поток, когда используются вместе с операторами ввода-вывода. Совсем не обязательно, что при этом какие-то данные считываются или записываются. Основные манипуляторы, определенные в заголовках `<istream>` и `<ostream>`, представлены в табл. 15.8.

[1] Эти классы заменяют функции-члены, которые ранее использовались для реализации библиотеки IOStream (`ipfx()`, `isfx()`, `opfx()` и `osfx()`). Эти классы гарантируют, что заключительная обработка будет выполнена даже в том случае, если ввод-вывод был прекращен аварийно с генерацией исключения.

Таблица 15.8. Манипуляторы, определенные в заголовках <istream> и <ostream>

Манипулятор	Класс	Описание
endl	basic_ostream	Вставляет в буфер символ перехода на новую строку и выгружает содержимое буфера вывода на соответствующее устройство
ends	basic_ostream	Вставляет в буфер (завершающий) нулевой символ
flush	basic_ostream	Выгружает содержимое буфера вывода на соответствующее устройство
ws	basic_istream	Читает данные, игнорируя пробельные символы

Манипуляторы с аргументами

Некоторые манипуляторы имеют аргументы. Например, следующее выражение устанавливает минимальную ширину поля и символ-заполнитель для следующего оператора вывода:

```
std::cout << std::setw(6) << std::setfill('_');
```

Стандартные манипуляторы с аргументами определены в заголовочном файле <iomanip>, который необходимо включать в программу для работы с манипуляторами, имеющими аргументы.

```
#include <iomanip>
```

Стандартные манипуляторы, имеющие аргументы, влияют на форматирование, задавая общие установки формата (см. раздел 15.7), представление времени (см. раздел 16.4.3) и денежных символов (см. раздел 16.4.2).

15.6.1. Обзор манипуляторов

В табл. 15.9 приведен обзор всех манипуляторов, предусмотренных в стандартной библиотеке C++. Манипуляторы hexfloat, defaultfloat, put_time(), get_time(), put_money() и get_money() стали доступными в стандарте C++11.

Таблица 15.9. Манипуляторы из стандартной библиотеки C++

Манипулятор	Описание
endl	Записывает символ перехода на новую строку и выгружает содержимое буфера
ends	Записывает (завершающий) нулевой символ
flush	Выгружает содержимое буфера
ws	Читает данные, игнорируя пробельные символы
skipws	Игнорирует ведущие пробельные символы в случае применения оператора >>
noskipws	Не игнорирует ведущие пробельные символы в случае применения оператора >>
unitbuf	Выгружает содержимое буфера после выполнения каждого оператора записи

Окончание табл. 15.9

Манипулятор	Описание
`nounitbuf`	Не выгружает содержимое буфера после выполнения каждого оператора записи
`setiosflags(`*flags*`)`	Устанавливает маску *flags* как флаги форматирования
`resetiosflags(`*m*`)`	Сбрасывает все флаги группы, заданной маской *m*
`setw(`*val*`)`	Устанавливает ширину поля для следующего оператора ввода или вывода равной *val*
`setfill(`*c*`)`	Определяет аргумент *c* как символ-заполнитель
`left`	Выравнивание по левому краю
`right`	Выравнивание по правому краю
`internal`	Выравнивание знаков по левому краю, а значений — по правому
`boolalpha`	Текстовое представление булевых значений
`noboolalpha`	Числовое представление булевых значений
`showpos`	Вывод знака + для положительных чисел
`noshowpos`	Подавление знака + для положительных чисел
`uppercase`	Вывод букв шестнадцатеричного представления (и степени в экспоненциальном представлении) в верхнем регистре
`nouppercase`	Вывод букв шестнадцатеричного представления (и степени в экспоненциальном представлении) в нижнем регистре
`oct`	Чтение и запись целых восьмеричных чисел
`dec`	Чтение и запись целых десятичных чисел
`hex`	Чтение и запись целых шестнадцатеричных чисел
`showbase`	Указывать основу системы счисления для числовых значений
`noshowbase`	Не указывать основу системы счисления для числовых значений
`showpoint`	Всегда записывать десятичную точку для чисел с плавающей точкой
`noshowpoint`	Не записывать десятичную точку для чисел с плавающей точкой
`setprecision(`*val*`)`	Задает аргумент *val* как новую точность для чисел с плавающей точкой
`fixed`	Использовать десятичную запись чисел с плавающей точкой
`scientific`	Использовать научный формат для чисел с плавающей точкой
`hexfloat`	Использовать шестнадцатеричный научный формат для чисел с плавающей точкой
`defaultfloat`	Использовать обычную запись чисел с плавающей точкой
`put_time(`*val*`,`*fmt*`)`	Записывать дату и время в соответствии с форматом *fmt*
`get_time(`*val*`,`*fmt*`)`	Читать дату и время в соответствии с форматом *fmt*
`put_money(`*val*`)`	Записывать денежные величины с использованием символа местной валюты
`put_money(`*val*`, `*intl*`)`	Записывать денежные величины с использованием символа валюты, заданного аргументом *intl*
`get_money(`*val*`)`	Читать денежные величины с использованием символа местной валюты
`get_money(`*val*`, `*intl*`)`	Читать денежные величины с использованием символа валюты, заданного аргументом *intl*

15.6.2. Как работают манипуляторы

Манипуляторы реализуются с помощью очень простого трюка, который не только позволяет легко управлять потоками, но и демонстрирует мощь механизма перегрузки функций. Манипуляторы — это всего лишь функции, передаваемые операторам ввода-вывода как аргументы. Затем оператор вызывает эти функции. Например, оператор вывода в классе `ostream` обычно перегружается следующим образом:

```
ostream& ostream::operator << ( ostream& (*op)(ostream&))
{
   // вызываем функцию, переданную как параметр и получающую поток как аргумент
   return (*op)(*this);
}
```

Аргумент `op` — это указатель на функцию, которая получает поток *ostream* в качестве аргумента и возвращает его обратно. Если второй операнд оператора `<<` является такой функцией, то она вызывается, получая в качестве аргумента первый операнд оператора `<<`.

Это звучит довольно запутанно, но на самом деле в этом нет ничего сложного. Разъясним это на примере. Манипулятор, т.е. функция, `endl()` для потока `ostream` реализуется примерно так:

```
std::ostream& std::endl (std::ostream& strm)
{
   // записываем символ перехода на новую строку
   strm.put('\n');

   // выгружаем буфер вывода
   strm.flush();

   // возвращаем strm, чтобы разрешить образование цепочек
   return strm;
}
```

Этот манипулятор можно использовать в выражениях следующим образом:

```
std::cout << std::endl
```

Здесь оператор `<<` применяется к потоку `cout` с функцией `endl()` в качестве второго операнда. Реализация оператора `<<` преобразует это выполнение в вызов переданной функции с аргументом, представляющим собой поток.

```
std::endl(std::cout)
```

Того же самого эффекта можно достигнуть, выполнив это выражение непосредственно. Преимущество использования функции заключается в том, что в этом случае для манипулятора не обязательно указывать пространство имен.

```
endl(std::cout)
```

Это объясняется тем, что в соответствии с принципом *ADL-поиска* (*поиска, зависящего от аргумента, или поиска Кёнига*) поиск функций осуществляется в пространствах имен, в которых определены их аргументы, если их невозможно найти иначе.

Поскольку потоковые классы являются шаблонными классами, параметризованными символьным типом, настоящая реализация манипулятора `endl()` выглядит следующим образом:

```
template <typename charT, typename traits>
std::basic_ostream<charT,traits>&
std::endl (std::basic_ostream<charT,traits>& strm)
{
   strm.put(strm.widen('\n'));
   strm.flush();
   return strm;
}
```

Функция-член `widen()` используется для преобразования символа перехода на новую строку в набор символов, используемый потоком в настоящее время. Более подробно это описывается в разделе 15.8.

Точный способ обработки аргументов манипулятора зависит от реализации, и не существует стандартного способа реализации пользовательских манипуляторов с аргументами (пример приведен в следующем разделе).

15.6.3. Пользовательские манипуляторы

Для определения собственного манипулятора необходимо написать функцию наподобие `endl()`. Например, следующая функция определяет манипулятор, игнорирующий все символы вплоть до конца строки:

```
// io/ignore1.hpp

#include <istream>
#include <limits>

template <typename charT, typename traits>
inline
std::basic_istream<charT,traits>&
ignoreLine (std::basic_istream<charT,traits>& strm)
{
   // пропускаем до конца строки
   strm.ignore(std::numeric_limits<std::streamsize>::max(),
               strm.widen('\n'));

   // возвращаем поток для конкатенации
   return strm;
}
```

Этот манипулятор делегирует работу функции `ignore()`, игнорирующей все символы до конца строки (функция `ignore()` описана в разделе 15.5.1).

Применение этого манипулятора очень простое:

```
// игнорируем оставшуюся часть строки
std::cin >> ignoreLine;
```

Применяя этот манипулятор несколько раз, можно пропустить несколько строк:

```
// игнорируем две строки
std::cin >> ignoreLine >> ignoreLine;
```

Этот механизм работает, потому что вызов функции `ignore(max,c)` игнорирует все символы, пока в потоке ввода не будет обнаружен символ `c`, не будет считано `max`

символов или не будет достигнут конец потока. Однако, прежде чем функция вернет значение, этот символ также будет проигнорирован.

Как уже было сказано, существует несколько способов определить собственный манипулятор с аргументами. Например, следующий код игнорирует n строк:

```
// io/ignore2.hpp

#include <istream>
#include <limits>
class ignoreLine
{
   private:
      int num;
   public:
      explicit ignoreLine (int n=1) : num(n) {
      }

      template <typename charT, typename traits>
      friend std::basic_istream<charT,traits>&
      operator>> (std::basic_istream<charT,traits>& strm,
                  const ignoreLine& ign)
      {
         // пропускаем символы, пока символ конца строки
         // не будет обнаружен num раз
         for (int i=0; i<ign.num; ++i) {
             strm.ignore(std::numeric_limits<std::streamsize>::max(),
                         strm.widen('\n'));
      }

      // возвращаем поток для конкатенации
      return strm;
   }
};
```

Здесь манипулятор ignoreLine — это класс, получающий аргумент для инициализации, а оператор ввода перегружен для объектов этого класса.

15.7. Форматирование

Определение форматов ввода-вывода зависит от двух механизмов. Наиболее очевидным является механизм флагов форматирования, определяющих, например, точность представления чисел, символ-заполнитель и основу системы счисления. Кроме него существует возможность настраивать форматы с учетом национальных стандартов. В этом разделе рассматриваются флаги форматирования, а аспекты интернационального форматирования будут рассмотрены в разделе 15.8 и в главе 16.

15.7.1. Флаги форматирования

Классы ios_base и basic_ios<> имеют несколько членов, которые используются для определения разных форматов ввода-вывода. Например, некоторые члены хранят минимальную ширину поля или точность представления числа или символ-заполнитель.

Член типа `ios::fmtflags` хранит конфигурацию флагов, определяющих, например, ставить ли перед положительным числом знак + или нет, а также представлять ли булево значение в числовом или текстовом виде.

Некоторые из флагов форматирования образуют группы. Например, флаги для восьмеричного, десятичного и шестнадцатеричного форматов целых чисел образуют группу. Для облегчения работы с такими группами определяются специальные маски.

Таблица 15.10. Функции-члены для доступа к флагам форматирования

Функция-член	Описание
`setf` (*flags*)	Устанавливает *flags* как дополнительные флаги и возвращает предыдущее состояние всех флагов
`setf` (*flags, grp*)	Устанавливает *flags* как новые флаги группы, идентифицированной аргументом *grp*, и возвращает предыдущее состояние всех флагов
`unsetf` (*flags*)	Сбрасывает все флаги в маске *flags*
`flags()`	Возвращает все установленные флаги форматирования
`flags` (*flags*)	Устанавливает *flags* как новые флаги и возвращает предыдущее состояние всех флагов
`copyfmt` (*stream*)	Копирует *все* определения форматов из аргумента *stream*

Некоторые функции-члены можно использовать для обработки всех определений формата в потоке (см. табл. 15.10). Функции `setf()` и `unsetf()` устанавливают и сбрасывают соответственно один или несколько флагов. Несколькими флагами можно манипулировать как одним целым, объединив их оператором "порязрядное или", т.е. оператором `|`. Функция `setf()` может получать маску как второй аргумент, чтобы сбросить все флаги в группе до того, как она установит флаги первого аргумента, которые также образуют группу. Версия функции `setf()` с одним аргументом этого не позволяет. Рассмотрим пример:

```
// устанавливаем флаги showpos и uppercase
std::cout.setf (std::ios::showpos | std::ios::uppercase);

// устанавливаем только флаг hex в группе basefield
std::cout.setf (std::ios::hex, std::ios::basefield);

// сбрасываем флаг uppercase
std::cout.unsetf (std::ios::uppercase);
```

С помощью функции `flags()` со всеми флагами форматирования можно работать как с единым целым. Если функция `flags()` вызывается без аргументов, она возвращает текущие флаги форматирования. Если функция `flags()` вызывается с аргументом, она извлекает из него новое состояние всех флагов форматирования и возвращает старое состояние. Таким образом, функция `flags()` с аргументом сбрасывает все флаги и устанавливает переданные флаги. Функция `flags()` оказывается полезной, например, для сохранения текущего состояния флагов, чтобы впоследствии восстановить предыдущее состояние. Эта возможность иллюстрируется следующим примером:

```
using std::ios;
using std::cout;

// сохраняет текущие флаги форматирования
ios::fmtflags oldFlags = cout.flags();
```

```
// вносим некоторые изменения
cout.setf(ios::showpos | ios::showbase | ios::uppercase);
cout.setf(ios::internal, ios::adjustfield);
cout << std::hex << x << std::endl;

// восстанавливаем сохраненные флаги форматирования
cout.flags(oldFlags);
```

С помощью функции `copyfmt()` можно копировать всю информацию о форматировании из одного потока в другой. Пример приведен в разделе 15.11.1. Для установки и сброса флагов форматирования можно также использовать манипуляторы (табл. 15.11).

Таблица 15.11. Манипуляторы для доступа к флагам форматирования

Манипулятор	Описание
`setiosflags(`*flags*`)`	Устанавливает флаги форматирования (вызывает `setf(`*flags*`)` для потоков)
`resetiosflags(`*mask*`)`	Сбрасывает все флаги группы, заданной маской (вызывает `setf(`*0,mask*`)` для потока)

Манипуляторы `setiosflags()` и `resetiosflags()` позволяют устанавливать и сбрасывать соответственно один или несколько флагов при выполнении операторов чтения << и записи >>.

Для использования одного из этих манипуляторов необходимо включить в программу заголовочный файл `<iomanip>`. Например:

```
#include <iostream>
#include <iomanip>
...
std::cout << resetiosflags(std::ios::adjustfield) // сбрасываем флаги adjustm
          << setiosflags(std::ios::left);         // выравнивание по левому краю
```

Некоторые манипуляции с флагами выполняются с помощью специальных манипуляторов. Они часто используются благодаря своему удобству и читабельности. Связанные с ними вопросы обсуждаются в следующих подразделах.

15.7.2. Формат ввода-вывода булевых значений

Флаг `boolalpha` определяет формат, используемый для чтения и записи булевых значений. Он указывает вид представления булевых значений — числовой или текстовый (табл. 15.12).

Таблица 15.12. Флаг для представления булевых значений

Флаг	Описание
`boolalpha`	Если флаг установлен, то используется текстовое представление, в противном случае — числовое

Если этот флаг не установлен (по умолчанию), то булевы значения выводятся как числовые строки. В данном случае для представления значения `false` всегда используется `0`, а для `true` — `1`. Если при чтении булевого значения как числовой строки ее значение

отличается от 0 или 1, то это считается ошибкой (и сопровождается установкой флага `failbit` для потока).

Если флаг установлен, то булевы значения записываются в текстовом виде. При чтении булева значения строка должна быть равной текстовому представлению `true` или `false`. Для определения текстовых строк, представляющих значения `true` и `false`, используется объект локального контекста потока (см. разделы 15.8 и 16.2.2). Стандартный объект локального контекста `"C"` для представления булевых значений использует строки `"true"` и `"false"`.

Для удобного манипулирования этим флагом предусмотрены специальные манипуляторы (табл. 15.13).

Таблица 15.13. Манипуляторы для булевого представления

Манипулятор	Описание
`boolalpha`	Устанавливает текстовое представление (устанавливает флаг `ios::boolalpha`)
`noboolalpha`	Устанавливает числовое представление (сбрасывает флаг `ios::boolalpha`)

Например, следующий код выводит переменную `b` сначала в числовом, а потом в текстовом представлении:

```
bool b;
...
std::cout << std::noboolalpha << b << " == "
          << std::boolalpha << b << std::endl;
```

15.7.3. Ширина поля, символ-заполнитель и выравнивание

Для определения ширины поля и символа заполнителя используются две функции-члена: `width()` и `fill()` (табл. 15.14).

Таблица 15.14. Функции-члены для установки ширины поля и символа-заполнителя

Функция-член	Описание
`width()`	Возвращает текущую ширину поля
`width(`*val*`)`	Устанавливает ширину поля для следующего форматированного вывода в аргумент *val* и возвращает предыдущую ширину поля
`fill()`	Возвращает текущий символ-заполнитель
`fill(`*c*`)`	Определяет аргумент *c* как символ-заполнитель и возвращает предыдущий символ-заполнитель

Использование ширины поля, символа-заполнителя и выравнивания для вывода

Функция `width()` определяет минимальную ширину поля при выводе. Это определение относится только к следующему форматированному полю, предназначенному для вывода. Если функция `width()` вызывается без аргументов, то она возвращает текущую ширину поля. Если функция `width()` вызывается с целочисленным аргументом, то она

изменяет ширину поля и возвращает предыдущую ширину. По умолчанию минимальная ширина равна 0. Это означает, что размер поля может быть произвольным. Кроме того, это же значение устанавливается после вывода значения.

Ширина поля не может использоваться для усечения вывода. Иначе говоря, максимальную ширину поля задать невозможно. Вместо этого ее придется программировать самостоятельно. Например, можно записать данные в строку и выводить только определенное количество символов.

Функция-член `fill()` определяет символ-заполнитель, который заполняет промежутки между форматированным представлением величины и позицией, соответствующей минимальной ширине поля. По умолчанию заполнителем является пробел.

Выравнивание значений в поле определяется тремя флагами (табл. 15.15), которые определены в классе `ios_base` вместе с соответствующей маской.

Таблица 15.15. Маски для выравнивания значений в поле

Маска	Флаг	Описание
`adjustfield`	`left`	Выравнивание значения по левому краю
	`right`	Выравнивание значения по правому краю
	`internal`	Выравнивание знака по левому краю, а значений — по правому
	Нет	Выравнивание значения по правому краю (по умолчанию)

В табл. 15.16 представлены результаты работы функций и флаги для разных значений. В качестве символа-заполнителя используется символ подчеркивания.

Таблица 15.16. Примеры выравнивания

Выравнивание	width()	-42	0.12	«Q»	'Q'
`left`	6	`-42___`	`0.12__`	`Q_____`	`Q_____`
`right`	6	`___-42`	`__0.12`	`_____Q`	`_____Q`
`internal`	6	`-___42`	`__0.12`	`_____Q`	`_____Q`

После выполнения любой операции форматированного вывода ширина поля, заданная по умолчанию, восстанавливается. Значения символа-заполнителя и установки выравнивания остаются неизменными, пока не будут изменены явным образом.

Для работы с шириной поля, символом-заполнителем и установками выравнивания предусмотрено несколько манипуляторов (табл. 15.17).

Манипуляторы `setw()` и `setfill()` используют аргумент, поэтому для работы с ними в программу необходимо включить заголовок `<iomanip>`. Например, код

```
#include <iostream>
#include <iomanip>
...
std::cout << std::setw(8) << std::setfill('_') << -3.14
          << ' ' << 42 << std::endl;
std::cout << std::setw(8) << "sum: "
          << std::setw(8) << 42 << std::endl;
```

выводит следующие строки:

```
___-3.14 42
___sum: _____42
```

Таблица 15.17. Манипуляторы для выравнивания

Манипулятор	Описание
`setw(`*val*`)`	Устанавливает ширину поля для следующей операции ввода или вывода в поток *val* (соответствует функции `width()`)
`setfill(`*c*`)`	Определяет символ *c* как символ-заполнитель (соответствует функции `fill()`)
`left`	Выравнивание значения по левому краю
`right`	Выравнивание значения по правому краю
`internal`	Выравнивание знака по левому краю, а значения — по правому

Использование ширины поля для ввода

Ширину поля можно использовать для определения максимального количества считываемых символов при чтении символьных последовательностей типа `char*`. Если `width()` не равно `0`, то считываются не более `width()-1` символов.

Поскольку обычные С-строки не могут увеличиваться по мере считывания, при их чтении с помощью операции >> необходимо всегда использовать функции `width()` или `setw()`. Например:

```
char buffer[81];
// читаем не более 80 символов:
cin >> setw(sizeof(buffer)) >> buffer;
```

Этот фрагмент кода считывает не более `80` символов, хотя `sizeof(buffer)` равно `81`, потому что один символ используется как (завершающий) нулевой символ, который добавляется автоматически. Отметим, что следующий ввод вызывает типичную ошибку:

```
char* s;
cin >> setw(sizeof(s)) >> s; // ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ
```

Причина заключается в том, что `s` объявлен как указатель, а не хранилище символов, а `sizeof(s)` — это размер указателя, а не хранилища, на которое он ссылается. Это типичная проблема, возникающая при работе с С-строками. Используя строки, можно избежать подобных проблем:

```
string buffer;
cin >> buffer; // OK
```

15.7.4. Положительный знак и верхний регистр

Для влияния на общий вид числовых значений используются два флага: `showpos` и `uppercase` (табл. 15.18).

Таблица 15.18. Флаги, влияющие на знак и символы для числовых значений

Флаг	Описание
`showpos`	Записывать знак + при выводе положительных значений
`uppercase`	Использовать верхний регистр при выводе букв

Если флаг ios::showpos установлен, то при выводе положительных чисел следует выводить знак +. Если этот флаг не установлен, то знак будет выводиться только при выводе отрицательных чисел. Если установлен флаг ios::uppercase, то буквы при выводе числовых значений будут выводиться в верхнем регистре. Этот флаг применяется к целым числам, которые выводятся в шестнадцатеричном и научном формате. По умолчанию буквы выводятся в нижнем регистре, а знак + не выводится. По умолчанию операторы

```
std::cout << 12345678.9 << std::endl;
std::cout.setf (std::ios::showpos | std::ios::uppercase);
std::cout << 12345678.9 << std::endl;
```

выводят следующие строки:

```
1.23457e+07
+1.23457E+07
```

Оба флага можно установить или сбросить с помощью манипуляторов, указанных в табл. 15.19.

Таблица 15.19. Манипуляторы для знака и букв в представлении числовых значений

Манипулятор	Описание
showpos	Выводить знак + при выводе положительных чисел (устанавливает флаг ios::showpos)
noshowpos	Не выводить знак + при выводе положительных чисел (сбрасывает флаг ios::showpos)
uppercase	Выводить буквы в верхнем регистре (устанавливает флаг ios::uppercase)
nouppercase	Выводить буквы в нижнем регистре (сбрасывает флаг ios::uppercase)

15.7.5. Основание системы счисления

Для выбора основания системы счисления при вводе-выводе используется группа, состоящая из трех флагов, которые определены в классе ios_base вместе с соответствующей маской (табл. 15.20).

Таблица 15.20. Флаги, задающие основание системы счисления при выводе целых чисел

Маска	Флаг	Описание
basefield	oct	Выводить и вводить восьмеричные числа
	dec	Выводить и вводить десятичные числа (по умолчанию)
	hex	Выводить и вводить шестнадцатеричные числа
	Нет	Выводить десятичные числа, а вводить в соответствии с начальными символами целочисленного значения

Изменение основания системы счисления распространяется на вывод всех целых чисел, пока флаги не будут изменены. По умолчанию используется десятичный формат. Двоичная система счисления не поддерживается. Однако существует возможность вводить и выводить целочисленные значения с помощью класса `bitset` (см. раздел 12.5.1).

Если ни один из флагов основания не установлен или установлено сразу несколько флагов, то при выводе используется десятичная система счисления.

Флаги основания системы счисления влияют и на ввод. Если установлен один из флагов основания, то все числа считываются именно в этой системе счисления. Если ни один из флагов основания не установлен, то система счисления определяется по начальным символам: число, начинающееся с `0x` или `0X`, вводится как шестнадцатеричное. Число, начинающееся с `0`, вводится как восьмеричное. Во всех остальных случаях число вводится как десятичное.

Существуют два способа изменить эти флаги.

1. Сбросить один флаг и установить другой:

```
std::cout.unsetf (std::ios::dec);
std::cout.setf (std::ios::hex);
```

2. Установить один флаг и автоматически сбросить все остальные флаги группы:

```
std::cout.setf (std::ios::hex, std::ios::basefield);
```

Кроме того, стандартная библиотека C++ содержит манипуляторы, значительно упрощающие работу с флагами (табл. 15.21).

Таблица 15.21. Манипуляторы, определяющие основу системы счисления

Манипулятор	Описание
`oct`	Вывод и ввод восьмеричных чисел
`dec`	Вывод и ввод десятичных чисел
`hex`	Вывод и ввод шестнадцатеричных чисел

Например, следующие инструкции выводят переменные `x` и `y` в шестнадцатеричном виде, а переменную `z` в десятичном:

```
int x, y, z;
...
std::cout << std::hex << x << std::endl;
std::cout << y << ' ' << std::dec << z << std::endl;
```

Дополнительный флаг `showbase` позволяет выводить числа в соответствии с обычными правилами C/C++, принятыми для индикации основания системы счисления при выводе литералов (табл. 15.22).

Таблица 15.22. Флаги для индикации основания системы счисления

Флаг	Описание
`showbase`	Если установлен, то при выводе указывается основание системы счисления

Если флаг ios::showbase установлен, то перед восьмеричными числами выводится префикс 0, а перед шестнадцатеричными — префикс 0x или, если установлен флаг ios::uppercase, префикс 0X. Например, код

```
std::cout << 127 << ' ' << 255 << std::endl;

std::cout << std::hex << 127 << ' ' << 255 << std::endl;

std::cout.setf(std::ios::showbase);
std::cout << 127 << ' ' << 255 << std::endl;

std::cout.setf(std::ios::uppercase);
std::cout << 127 << ' ' << 255 << std::endl;
```

выводит результат

```
127 255
7f ff
0x7f 0xff
0X7F 0XFF
```

Флагом ios::showbase можно также управлять с помощью манипуляторов, приведенных в табл. 15.23.

Таблица 15.23. Манипуляторы для индикации основания системы счисления

Манипулятор	Описание
showbase	Выводить основание системы счисления (устанавливает флаг ios::showbase)
noshowbase	Не выводить основание системы счисления (сбрасывает флаг ios::showbase)

15.7.6. Вывод чисел с плавающей точкой

Несколько флагов и функций-членов управляют выводом чисел с плавающей точкой. Флаги, представленные в табл. 15.24, определяют вид, в котором следует выводить числа: с десятичной точкой или в научном формате. Эти флаги определены в классе ios_base вместе с соответствующей маской. Если установлен флаг ios::fixed, то числа выводятся с десятичной точкой. Если установлен флаг ios::scientific, то используется научное или экспоненциальное представление.

Таблица 15.24. Флаги для вывода чисел с плавающей точкой

Маска	Флаг(и)	Описание
floatfield	fixed	Использовать десятичное представление
	scientific	Использовать научное представление
	Нет	Использовать "лучшее" из двух представлений (по умолчанию)
	fixed|scientific	Использовать шестнадцатеричное научное представление (по стандарту C++11)

До принятия стандарта C++11 установка флагов `fixed|scientific` не была определена. После принятия стандарта C++11 их можно использовать для определения шестнадцатеричного научного представления, а также спецификатор формата `%a` для функции `printf()` для вывода шестнадцатеричного значения в степени `2`. Например, число `234.5` записывается в виде `0x1.d5p+7` (`0x1.d5` умножить на 2^7, т.е. `1*128/1 + 13*128/16 + 5*128/256`).

С помощью флага `showpoint` можно заставить поток записывать десятичную точку и замыкающие нули в соответствии с текущей точностью (табл. 15.25).

Таблица 15.25. Флаг для вывода десятичной точки

Флаг	Описание
`showpoint`	Всегда выводить десятичную точку и заполнять позиции замыкающими нулями

Для определения точности предусмотрена функция-член `precision()` (см. табл. 15.26).

Таблица 15.26. Функция-член для задания точности чисел с плавающей точкой

Функция-член	Описание
`Функция: форматирования: precision()`	Возвращает текущую точность представления числа с плавающей точкой
`precision(`*val*`)`	Аргумент *val* задает новую точность представления числа с плавающей точкой и возвращает старую

Если используется научное представление, то функция `precision()` определяет количество десятичных знаков в дробной части. Во всех случаях остаток не усекается, а округляется. Если функция `precision()` вызывается без аргументов, то она возвращает текущую точность. Если эта функция вызывается с аргументом, то она задает точность в соответствии с этим аргументом и возвращает предыдущую точность. По умолчанию точность составляет шесть десятичных знаков.

По умолчанию флаги `ios::fixed` и `ios::scientific` не установлены. В этом случае представление числа зависит от выводимого значения и при этом выводится не более `precision()` десятичных знаков, так что начальный нуль перед десятичной точкой и/или все завершающие нули и, возможно, даже десятичная точка удаляются. Если точность `precision()` является достаточной, то используется десятичное представление, в противном случае — научное.

В табл. 15.27 приведены довольно сложные зависимости между флагами и точности, где в качестве примера используются два конкретных значения.

Таблица 15.27. Пример форматирования чисел с плавающей точкой

	precision()	421.0	0,0123456789
Обычное	2	`4.2e+02`	`0.012`
	6	`421`	`0.0123457`
С точкой	2	`4.2e+02`	`0.012`
	6	`421.000`	`0.0123457`
`fixed`	2	`421.00`	`0.01`
	6	`421.000000`	`0.012346`

Окончание табл. 15.27

	precision()	421.0	0,0123456789
scientific	2	4.21e+02	1.23e-02
	6	4.210000e+02	1.234568e-02
fixed|scientific	2	0x1.a5p+8	0x1.95p-7
	6	0x1.a50000p+8	0x1.948b10p-7

Для того чтобы перед положительными целыми числами выводился знак +, можно использовать флаг `ios::showpos`, а флаг `ios::uppercase` можно использовать для того, чтобы определить регистр, в котором будут использоваться буквы научного представления: нижний или верхний.

Флаг `ios::showpoint`, представление и точность можно задавать с помощью манипуляторов, представленных в табл. 15.28.

Таблица 15.28. Манипуляторы для чисел с плавающей точкой

Манипулятор	Описание
`showpoint`	Всегда записывать десятичную точку (устанавливает флаг `ios::showpoint`)
`noshowpoint`	Не выводит десятичную точку (сбрасывает флаг `showpoint`)
`setprecision(`*val*`)`	Устанавливает *val* как новое значение точности
`fixed`	Использовать десятичное представление
`scientific`	Использовать научное представление
`hexfloat`	Использовать шестнадцатеричное научное представление (начиная со стандарта C++11)
`defaultfloat`	Использовать обычное представление (сбрасывает флаг `floatfield`; начиная со стандарта C++11)

Например, инструкция

```
std::cout << std::scientific << std::showpoint
          << std::setprecision(8)
          << 0.123456789 << std::endl;
```

выводит результат

```
1.23456789e-01
```

Функция `setprecision()` представляет собой манипулятор с аргументом, поэтому для ее использования необходимо включать заголовочный файл `<iomanip>`.

15.7.7. Общие определения формата

Список флагов форматирования завершают еще два флага: `skipws` и `unitbuf` (табл. 15.29).

По умолчанию флаг `ios::skipws` установлен, т.е. оператор >> игнорирует ведущие пробельные символы. Это полезно во многих ситуациях. Например, если этот флаг установлен, то явно считывать разделяющие пробелы между числами не обязательно. Однако из этого следует, что чтение пробелов с помощью оператора >> невозможно, потому что ведущие пробельные символы всегда будут игнорироваться.

Таблица 15.29. Другие флаги форматирования

Флаг	Описание
`skipws`	Автоматически игнорировать начальные пропуски при вводе значения с помощью оператора >>
`unitbuf`	Выгружать содержимое буфера после каждой операции вывода

С помощью флага `ios::unitbuf` осуществляется буферизация вывода. Если он установлен, то вывод не буферизуется, т.е. после каждой операции вывода содержимое буфера выгружается. По умолчанию для большинства потоков этот флаг не установлен. Однако для потоков `cerr` и `wcerr` этот флаг установлен изначально.

Обоими флагами можно управлять с помощью манипуляторов, представленных в табл. 15.30.

Таблица 15.30. Манипуляторы для других флагов форматирования

Манипулятор	Описание
`skipws`	Игнорировать ведущие пробельные символы при выполнении оператора >> (устанавливает флаг `ios::skipws`)
`noskipws`	Не игнорировать начальные пропуски при выполнении оператора >> (сбрасывает `ios::skipws`)
`unitbuf`	Выгружать содержимое буфера после каждой операции вывода (устанавливает флаг `ios::unitbuf`)
`nounitbuf`	Не выгружать содержимое буфера после каждой операции вывода (сбрасывает флаг `ios::unitbuf`)

15.8. Интернационализация

Форматы ввода-вывода можно адаптировать к национальным стандартам. Для этого в классе `ios_base` определены функции-члены, приведенные в табл. 15.31.

Таблица 15.31. Функции-члены для интернационализации

Функция-член	Описание
`imbue(`*loc*`)`	Задает объект локального контекста
`getloc()`	Возвращает текущий объект локального контекста

Каждый поток использует связанный с ним объект локального контекста. По умолчанию начальным объектом локального контекста является копия глобального объекта локального контекста на момент создания потока. Объект локального контекста определяет, например, детали форматирования чисел, такие как символ десятичной точки или строки для представления булевых значений.

В противоположность средствам локализации из языка C в языке C++ каждый поток можно конфигурировать отдельным объектом локального контекста. Это позволяет, например, вводить числа с плавающей точкой в соответствии с американским стандартом, а выводить — в немецком (в немецком стандарте для представления десятичной точки используется запятая). Примеры и подробное описание приведены в разделе 16.2.1.

В кодировке потока необходимы специальные символы. По этой причине для потоков предусмотрено несколько функций преобразования (табл. 15.32).

Таблица 15.32. Функции потоков для интернационализации символов

Функция-член	Описание
`widen(c)`	Преобразует символ *c* типа `char` в символ в кодировке потока
`narrow(c,def)`	Преобразует символ *c* из кодировки потока в тип `char`; если соответствующего символа типа `char` не существует, то возвращается аргумент *def*

Например, чтобы получить символ перехода на новую строку из кодировки потока `strm`, можно выполнить вызов

```
strm.widen('\n')
```

Дополнительные сведения, относящиеся к объектам локального контекста и вопросам интернационализации, приведены в главе 16.

15.9. Доступ к файлам

Потоки можно использовать для доступа к файлам. В этом разделе обсуждаются соответствующие функциональные возможности.

15.9.1. Классы файловых потоков

Стандартная библиотека C++ содержит четыре шаблонных класса, для которых заранее определены следующие стандартные специализации.

1. Шаблонный класс `basic_ifstream<>` со специализациями `ifstream` и `wifstream`, обеспечивающими чтение файлов ("файловый поток ввода").
2. Шаблонный класс `basic_ofstream<>` со специализациями `ofstream` и `wofstream`, обеспечивающими вывод в файлы ("файловый поток вывода").
3. Шаблонный класс `basic_fstream<>` со специализациями `fstream` и `wfstream`, обеспечивающими чтение и запись файлов.
4. Шаблонный класс `basic_filebuf<>` со специализациями `filebuf` и `wfilebuf`, которые используются другими классами файловых потоков для фактического чтения и записи символов.

Как показано на рис.15.2, эти классы связаны с базовыми классами потоков и объявлены в заголовочном файле `<fstream>` следующим образом:

```
namespace std {
   template <typename charT,
             typename traits = char_traits<charT> >
             class basic_ifstream;
   typedef basic_ifstream<char> ifstream;
   typedef basic_ifstream<wchar_t> wifstream;

   template <typename charT,
             typename traits = char_traits<charT> >
```

```
          class basic_ofstream;
   typedef basic_ofstream<char> ofstream;
   typedef basic_ofstream<wchar_t> wofstream;

   template <typename charT,
   typename traits = char_traits<charT> >
          class basic_fstream;
   typedef basic_fstream<char> fstream;
   typedef basic_fstream<wchar_t> wfstream;

   template <typename charT,
            typename traits = char_traits<charT> >
            class basic_filebuf;
   typedef basic_filebuf<char> filebuf;
   typedef basic_filebuf<wchar_t> wfilebuf;
}
```

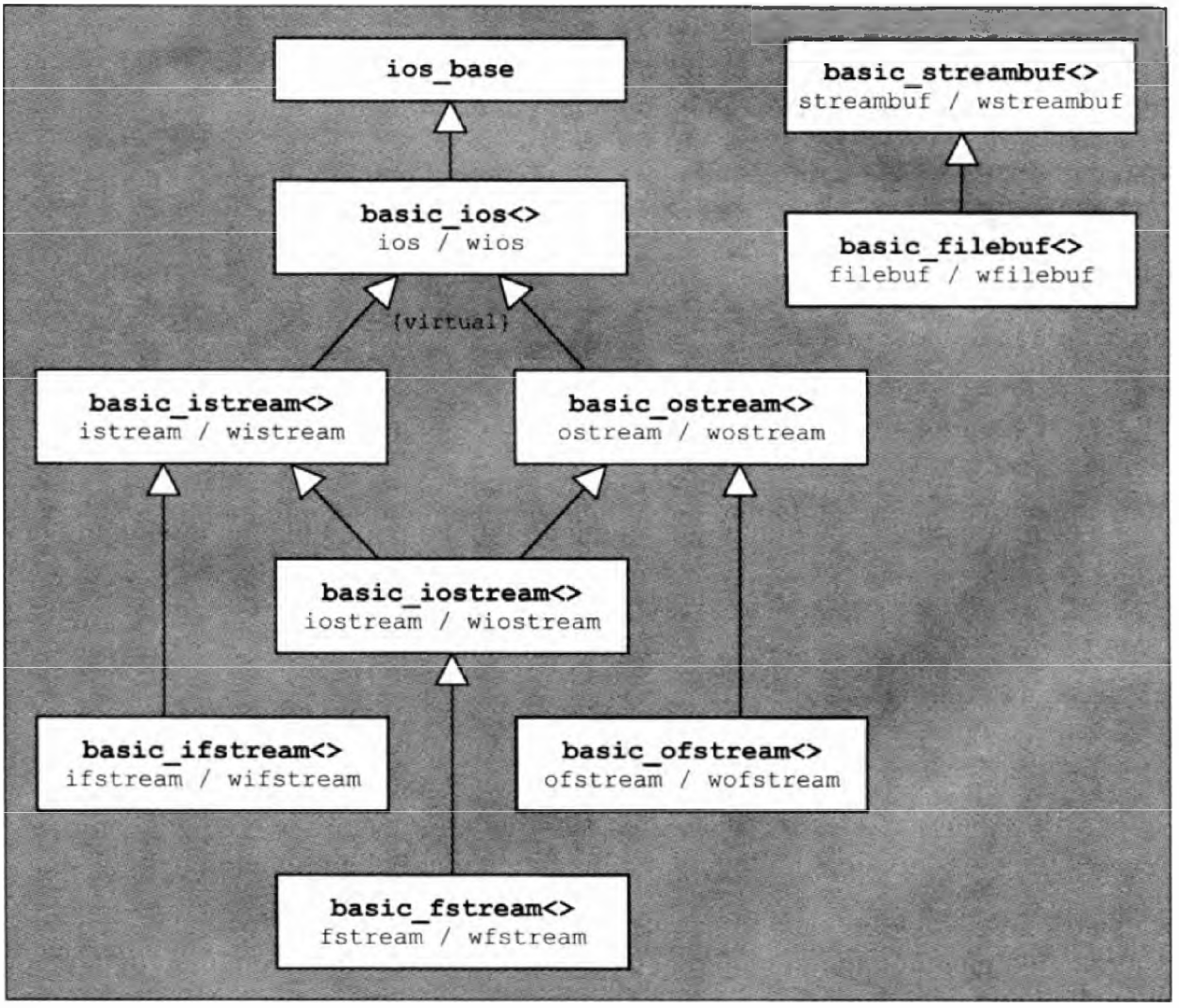


Рис. 15.2. Иерархия классов файловых потоков

По сравнению с механизмом языка С, основное преимущество классов файловых потоков заключается в автоматическом управлении файлами. Файлы автоматически открываются в момент создания потока и закрываются в момент его уничтожения. Разумеется, это возможно благодаря соответствующим конструкторам и деструкторам.

Важно помнить, что для потоков, которые и читаются, и записываются, *невозможно* произвольным образом переходить от чтения к записи, и наоборот[12]. Для того чтобы переключиться с чтения на запись файла, и наоборот, после начала чтения или записи файла необходимо выполнить операцию позиционирования, возможно, сохраняя текущую позицию. Единственным исключением из этого правила является чтение, при котором достигнут конец файла. В этом случае символы можно записывать немедленно. Нарушение этого правила может привести к самым разным странным последствиям.

Если объект файлового потока создается конструктором, получающим аргумент в виде строки или С-строки, то попытка открытия файла для чтения и/или записи выполняется автоматически. Если попытка оказалась успешной, то этот факт отражается на состоянии потока. Таким образом, состояние можно проверить после создания потока.

Следующая программа сначала открывает файл `charset.out`, выводит в файл текущий набор символов — все символы в диапазоне от 32 до 255 и выводит его содержимое:

```
// io/fstream1.cpp

#include <string>    // для строк
#include <iostream>  // для ввода-вывода
#include <fstream>   // для файлового ввода-вывода
#include <iomanip>   // для функции setw()
#include <cstdlib>   // для функции exit()
using namespace std;

// опережающие определения
void writeCharsetToFile (const string& filename);
void outputFile (const string& filename);

int main ()
{
   writeCharsetToFile("charset.out");
   outputFile("charset.out");
}

void writeCharsetToFile (const string& filename)
{
   // открываем файл для записи
   ofstream file(filename);

   // файл открыт?
   if (! file) {
       // НЕТ, прерываем выполнение программы
       cerr << "can't open output file \"" << filename << "\""
            << endl;
       exit(EXIT_FAILURE);
   }

   // выводим набор символов
   for (int i=32; i<256; ++i) {
        file << "value: " << setw(3) << i << " "
             << "char: " << static_cast<char>(i) << endl;
```

[12] Это ограничение унаследовано от языка С. Однако, вероятно, реализации стандартной библиотеки будут по-прежнему поддерживать его.

```
    }

} // файл закрывается автоматически

void outputFile (const string& filename)
{
    // открываем файл для чтения
    ifstream file(filename);

    // файл открыт?
    if (! file) {

    // НЕТ, прерываем выполнение программы
    cerr << "can't open input file \"" << filename << "\""
         << endl;
    exit(EXIT_FAILURE);
}

    // копируем содержимое файла в объект cout
    char c;
    while (file.get(c)) {
           cout.put(c);
    }
} // файл закрывается автоматически
```

В функции `writeCharsetToFile()` конструктор класса `ofstream` открывает файл с заданным именем

```
std::ofstream file(filename); // открываем файл для вывода
```

К сожалению, до принятия стандарта C++11 классы файловых потоков не имели конструктора, получающего аргумент класса `string`. Итак, до принятия стандарта C++11 необходимо было писать такой код:

```
std::ofstream file(filename.c_str()); // открытие файла
                                      // до принятия стандарта C++11
```

После этого объявления программа проверяет, находится ли поток в нормальном состоянии.

```
if (! file) {


    ...


}
```

Если открытие файла не было успешным, тест завершается ошибкой. После этой проверки в цикле выводятся значения от `32` до `255` с соответствующими символами.

В функции `outputFile()` конструктор класса `ifstream` открывает файл для чтения. Затем содержимое файла считывается и выводится символ за символом.

В конце обеих функций открытый файл закрывается автоматически, когда соответствующий поток выходит из области видимости. Если в момент уничтожения потока файл остается открытым, его закрывают деструкторы классов `ifstream` и `ofstream`.

Вместо копирования содержимого файла символ за символом можно вывести все содержимое с помощью одной инструкции, передавая указатель на буфер файлового потока оператора <<.

```
// копируем содержимое файла в объект cout
std::cout << file.rdbuf();
```

Подробности изложены в разделе 15.14.3.

15.9.2. Rvalue и семантика перемещения для файловых потоков

После принятия стандарта C++11 файловые потоки поддерживают rvalue и семантику перемещения. Фактически `ostream` реализуют операцию вывода, а потоки `istream` — операцию ввода. Это позволяет использовать временные объекты потоков. Например, можно осуществить вывод во временный файловый поток[13]:

```
// io/fstream2.cpp

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

int main()
{
   // выводим строку во временный файловый поток(по стандарту C++11)
   std::string s("hello");
   std::ofstream("fstream2.tmp") << s << std::endl;

   // выводим С-строку во временный файловый поток
   // - ПРМЕЧАНИЕ: до принятия стандарта C++1 записывался указатель
   std::ofstream("fstream2.tmp", std::ios::app) << "world" << std::endl;
}
```

Эта программа выводит строки `"hello"` и `"world"` в файл `"fstream2.tmp"` (начиная со стандарта C++11). До принятия стандарта C++11 вместо первой инструкции следовало написать следующий код:

```
std::string s("hello");
std::ofstream os("fstream2.tmp");
os << s << std::endl;
```

До принятия стандарта C++11 второй оператор вывода компилировался, но работал очень неожиданно: он выводил в файл `"fstream2.tmp"` *значение указателя*. Причина этого заключалась в том, что вызывалась соответствующая функция-член (см. раздел 15.3.3).

```
ostream& ostream::operator<< (const void* ptr);
```

Кроме того, файловые потоки теперь поддерживают семантику перемещения и обмена с помощью конструктора перемещения, операции перемещающего присваивания и функции

[13] Благодарю за этот пример Даниэля Крюглера (Daniel Krügler).

swap(). Таким образом, можно передать файловый поток как аргумент или вернуть файловый поток из функции. Кроме того, если файл используется за пределами области видимости, в которой он был создан, в соответствии со стандартом его можно вернуть (описание процесса возвращения значений и семантики перемещения см. в разделе 3.1.5).

```
std::ofstream openFile (const std::string& filename)
{
   std::ofstream file(filename);
   ...
   return file;
}

std::ofstream file;
file = openFile("abc.tmp"); // используется возвращенный файловый поток
                            // (по стандарту C++11)
file << "hello, world" << std::endl;
```

До принятия стандарта C++11 было необходимо (а сейчас можно в качестве альтернативы) размещать файловый объект в куче и удалять его, если он стал ненужным.

```
std::ofstream* filePtr = new std::ofstream("abc.tmp");
...
delete filePtr;
```

В этом случае лучше использовать классы интеллектуальных указателей (см. раздел 5.2).

15.9.3. Флаги файлов

В классе ios_base определены флаги, управляющие режимами работы с файлами (табл. 15.33). Эти флаги имеют тип openmode, представляющий собой битовую маску, аналогичную типу fmtflags.

Таблица 15.33. Флаги для открытия файлов

Флаг	Описание
in	Открыть для ввода (по умолчанию для ifstream)
out	Открыть для вывода (по умолчанию для ofstream)
app	Всегда добавлять в конец при выводе
ate	Установить курсор на конец файла после открытия ("at end")
trunc	Удалить предыдущее содержимое файла
binary	Не заменять специальные символы

Флаг binary конфигурирует поток так, чтобы подавить преобразование специальных символов или символьных последовательностей, например конца строки или конца файла. В операционных системах, таких как Windows и OS/2, конец строки в текстовом файле представлен двумя символами (CR и LF). В обычном текстовом режиме (когда флаг binary не установлен) при вводе и выводе символ перехода на новую строку заменяется двухсимвольной последовательностью, и наоборот. В бинарном режиме (когда флаг binary установлен) эти преобразования не выполняются.

Флаг `binary` необходимо использовать всегда, если файл не содержит специальных символов, а его содержимое обрабатывается как двоичные данные. Примером является копирование файла путем ввода символ за символом и их вывода без модификации. Если файл обрабатывается как текстовый, флаг устанавливать не следует, потому что при этом требуется специальная обработка перехода на новую строку. Например, переход на новую строку может кодироваться двумя символами.

Некоторые реализации предусматривают дополнительные флаги, такие как `nocreate` (открываемый файл должен существовать) и `noreplace` (файл не должен существовать). Однако эти флаги не являются стандартными и, следовательно, не гарантируют переносимости.

Эти флаги можно объединять с помощью оператора `|`. Результирующую маску `openmode` можно передавать конструктору как второй необязательный аргумент. Например, следующий оператор открывает файл для добавления текста в его конец:

```
std::ofstream file("abc.out", std::ios::out|std::ios::app);
```

В табл. 15.34 перечислены разные комбинации флагов в интерфейсе функции `fopen()` из языка C, предназначенной для открытия файлов. Комбинации флагов `binary` и `ate` не указаны. Установка флага `binary` означает, что к строке приписывается символ `b`, а установка флага `ate` означает переход в конец файла сразу после его открытия. Остальные комбинации флагов, не приведенные в таблице, такие как `trunc|app`, не разрешаются. До принятия стандарта C++11 флаги `app`, `in|app` и `in|out|app` не были определены.

Таблица 15.34. Описание режимов открытия в языке C++

Флаги `ios_base`	Описание	Обозначение режима в языке C
`in`	Читать (файл должен существовать)	`"r"`
`out`	Стереть и записать (создать файл при необходимости)	`"w"`
`out|trunc`	Стереть и записать (создать файл при необходимости)	`"w"`
`out|app`	Добавить (создать файл при необходимости)	`"a"`
`app`	Добавить (создать файл при необходимости)	`"a"`
`in|out`	Читать и записывать; исходная позиция расположена в начале (файл должен существовать)	`"r+"`
`in|out|trunc`	Стереть, читать и записать (создать файл при необходимости)	`"w+"`
`in|app`	Дописать в конец (создать файл при необходимости)	`"a+"`
`in|out|app`	Дописать в конец (создать файл при необходимости)	`"a+"`

Тот факт, что файл открыт для чтения и/или записи, не зависит от соответствующего потокового класса. Если не используется второй аргумент, то класс определяет только режим открытия по умолчанию. Это означает, что только файл, используемый классом `ifstream` или `ofstream`, может быть открыт для чтения *и* записи. Режим открытия передается соответствующему классу буфера потока, который открывает файл. Однако операции, возможные над объектом, определяются классом потока.

Файл, связанный с файловым потоком, может быть открыт или закрыт явным образом. Для этого определены три функции-члены (табл. 15.35), которые полезны в основном в тех ситуациях, когда файловый поток создается без инициализации.

Для демонстрации их использования рассмотрим следующий пример, в котором все файлы открываются с именами, заданными аргументами командной строки, а затем выводится все содержимое (это соответствует программе `cat` операционной системы UNIX).

Таблица 15.35. Функции-члены для открытия и закрытия файлов

Функция-член	Описание
`open(`*name*`)`	Открывает файл для потока, используя режим открытия по умолчанию
`open(`*name*`,`*flags*`)`	Открывает файл для потока, используя режим открытия, заданный файлами
`close()`	Закрывает файл
`is_open()`	Проверяет, открыт ли файл

```
// io/cat1.cpp

// заголовочные файлы для ввода и вывода в файл
#include <fstream>
#include <iostream>
using namespace std;

// для всех имен файлов, переданных в виде аргументов командной строки
// - открыть, вывести на печать содержимое и закрыть файл
int main (int argc, char* argv[])
{
   ifstream file;

   // для всех аргументов командной строки
   for (int i=1; i<argc; ++i) {

   // открыть файл
   file.open(argv[i]);

   // вывести содержимое файла в поток cout
   char c;
   while (file.get(c)) {
          cout.put(c);
   }

   // сбросить флаги eofbit и failbit, установленные при обнаружении конца файла
   file.clear();

   // закрыть файл
   file.close();
   }
}
```

После обработки файла следует вызвать функцию `clear()`, чтобы сбросить флаги, установленные после обнаружения конца файла. Это необходимо потому, что объект

потока используется для нескольких файлов. Функция `open()` *никогда* не сбрасывает установленные флаги. Таким образом, если поток не находился в нормальном состоянии после закрытия и повторного открытия, необходимо снова вызвать функцию `clear()`, чтобы привести его в нормальное состояние. Это же касается открытия другого файла.

Вместо посимвольной обработки можно вывести все содержимое файла с помощью одной строки, передавая указатель на буфер файлового потока в виде аргумента оператора <<.

```
// вывести содержимое файла в объект cout
std::cout << file.rdbuf();
```

Подробности изложены в разделе 15.14.3.

15.9.4. Произвольный доступ

Таблица 15.36. Функции-члены для позиционирования в потоке

Класс	Функция-член	Описание
`basic_istream<>`	`tellg()`	Возвращает позицию ввода
	`seekg(`*pos*`)`	Устанавливает позицию ввода как абсолютное
	`seekg(`*offset*`,`*rpos*`)`	Устанавливает позицию ввода как относительное значение
`basic_ostream<>`	`tellp()`	Возвращает позицию вывода
	`seekp(`*pos*`)`	Устанавливает позицию ввода как абсолютное значение
	`seekp(`*offset*`,`*rpos*`)`	Устанавливает позицию ввода как относительное значение

В табл. 15.36 перечислены функции-члены, предназначенные для позиционирования в потоках C++. Эти функции различают позицию ввода и вывода (буква `g` означает *get* (ввести), а `p` — *put* (вывести)). Функции для управления позицией ввода определены в классе `basic_istream<>`, а функции для управления позицией вывода — в классе `basic_ostream<>`. Однако не все потоковые классы поддерживают позиционирование. Например, позиционирование потоков `cin`, `cout` и `cerr` не определено. Позиционирование файлов определено в базовых классах, потому что, как правило, используются ссылки на объекты типа `istream` и `ostream`.

Функции `seekg()` и `seekp()` можно вызывать как с абсолютными, так и с относительными позициями. Для работы с абсолютными значениями необходимо использовать функции `tellg()` и `tellp()`, возвращающие абсолютное значение типа `pos_type`. Это значение *не является* целочисленным или обычным индексом символа, потому что логическая и реальная позиции могут отличаться друг от друга. Например, в текстовых файлах Windows символ перехода на новую строку представляется в файлах двумя символами, даже если с логической точки зрения он представляет собой один символ. Ситуация еще более ухудшается, если для представления символов в файле используется многобайтовое представление символов.

Точное определение типа `pos_type` является немного сложным: в стандартной библиотеке C++ определен глобальный шаблонный класс `fpos<>` для позиционирования в файлах. Класс `fpos<>` используется для определения потока `streampos` символов типа `char`

и потока wstreampos символов типа wchar_t. Эти типы используются для определения типа pos_type соответствующих символьных свойств (см. раздел 16.1.4). Свойства типа pos_type используются для определения соответствующих потоковых классов. Таким образом, для представления позиции в потоке можно также использовать тип streampos. Однако использование типа long или unsigned long является ошибкой, потому что тип streampos больше *не* является целочисленным типом[14]. Например:

```
// сохраняем текущую позицию в файле
std::ios::pos_type pos = file.tellg();
...
// переходим к позиции, сохраненной в переменной pos
file.seekg(pos);
```

Вместо объявления

```
std::ios::pos_type pos;
```

можно было бы написать

```
std::streampos pos;
```

Для относительных значений смещение может определяться относительно трех позиций, для которых определены соответствующие константы (табл. 15.37). Эти константы определены в классе ios_base и имеют тип seekdir.

Таблица 15.37. Константы для относительных позиций

Константа	Описание
beg	Смещение задается относительно начала файла (сокращение слова "beginning")
cur	Смещение задается относительно текущей позиции (сокращение слова "current")
end	Смещение задается относительно конца файла

Смещение имеет тип off_type, который представляет собой косвенное определение типа streamoff. Аналогично типу pos_type, класс streamoff используется для определения типа свойств off_type (см. раздел 16.1.4) и потоковых классов. Однако streamoff — это целочисленный тип со знаком, поэтому для задания смещения можно использовать целые числа. Например:

```
// переходим в начало файла
file.seekg (0, std::ios::beg);
...
// переходим на 20 символов вперед
file.seekg (20, std::ios::cur);
...
// переходим на позицию за 10 символов до конца
file.seekg (-10, std::ios::end);
```

Во всех случаях позиция должна находиться в пределах файла. Если позиция предшествует началу или находится за концом файла, поведение программы не определено.

[14] Прежде тип streampos использовался для представления позиции в потоке и был определен как unsigned long.

Следующий пример демонстрирует использование функции `seekg()`. В нем используется функция, которая дважды выводит содержимое файла:

```
// io/cat2.cpp

// заголовочные файлы для файла ввода-вывода
#include <iostream>
#include <fstream>

void printFileTwice (const char* filename)
{
  // открываем файл
   std::ifstream file(filename);

  // выводим содержимое файла первый раз
   std::cout << file.rdbuf();

   // переходим в начало
   file.seekg(0);

   // выводим содержимое файла второй раз
   std::cout << file.rdbuf();
}

int main (int argc, char* argv[])
{
   // дважды выводим все файлы, передаваемые как аргумент командной строки
   for (int i=1; i<argc; ++i) {
        printFileTwice(argv[i]);
   }
}
```

Функция `file.rdbuf()` используется для вывода содержимого потока `file` (см. раздел 15.14.3). Таким образом, оператор применяется непосредственно к потоковому буферу и не может изменить состояние потока. Если бы содержимое файла `file` выводилось с помощью функций потокового интерфейса, например `getline()` (см. раздел 15.5.1), пришлось бы вызывать функцию `clear()`, чтобы очистить состояние потока `file` перед началом работы с ним (включая изменение позиции ввода), потому что эти функции после обнаружения конца файла устанавливают флаги `ios::eofbit` и `ios::failbit`.

Для манипуляции позициями ввода и вывода используются разные функции. Однако для стандартных потоков в одном и том же потоковом буфере для чтения и записи используется одна и та же позиция. Это важно, если несколько потоков используют один и тот же потоковый буфер (см. раздел 15.12.2).

15.9.5. Использование дескрипторов файлов

Некоторые реализации позволяют присоединить поток данных к ранее открытому каналу ввода-вывода. Для этого файловый поток данных инициализируется *файловым дескриптором*. Файловый дескриптор — это целое число, идентифицирующее открытый канал ввода-вывода. В операционных системах семейства UNIX файловые дескрипторы используются в низкоуровневом интерфейсе для работы с функциями ввода-вывода операционной системы. Определены три стандартных файловых дескриптора:

- 0 — стандартный канал ввода;
- 1 — стандартный канал вывода;
- 2 — стандартный канал вывода ошибок.

Эти каналы могут связываться с файлами, консолью, другими процессами или средствами ввода-вывода. К сожалению, стандартная библиотека C++ не поддерживает присоединение потоков данных к каналам ввода-вывода с помощью файловых дескрипторов, потому что язык не должен зависеть от конкретных особенностей операционных систем. Впрочем, на практике такая возможность существует, а ее единственный недостаток — снижение переносимости программ. На сегодняшний день в стандартах интерфейсов операционных систем, таких как POSIX или X/OPEN, не существует соответствующей спецификации. Несмотря на то что ее разработка пока не планируется, в стандарте C++11 уже зарезервировано пространство имен `posix`.

Тем не менее поток можно инициализировать с помощью файлового дескриптора. Описание и возможная реализация такой инициализации приведены в разделе 15.13.3.

15.10. Потоковые классы для работы со строками

Механизмы потоковых классов можно также использовать для чтения данных из строк и записи данных в строки. Строковые потоки имеют буфер, но не связаны с каналами ввода-вывода. Буфером и строкой можно манипулировать с помощью специальных функций. В основном эта возможность используется для обеспечения независимости фактического ввода-вывода. Например, текст для вывода можно форматировать в строке, а затем послать в канал вывода или прочитать данные строка за строкой и обработать их с помощью строковых потоков.

До появления стандарта C++98 классы строковых потоков использовали для представления строк тип `char*`. В настоящее время используется тип `string` (или, более широко, тип `basic_string<>`). Старые строковые потоковые классы также являются частью стандартной библиотеки C++, но объявлены нежелательными средствами. Таким образом, их не следует использовать в новых программах и необходимо постепенно заменять их в унаследованном коде. Тем не менее краткое описание этих классов по-прежнему приводится в конце этого раздела.

15.10.1. Строковые потоковые классы

Для строк определены следующие потоковые классы, аналогичные потоковым классам для файлов.

- Шаблонный класс `basic_istringstream<>` со специализациями `istringstream` и `wistringstream` для ввода данных из строк ("строковый поток ввода")
- Шаблонный класс `basic_ostringstream<>` со специализациями `ostringstream` и `wostringstream` для вывода данных в строки ("строковый поток вывода")
- Шаблонный класс `basic_stringstream<>` со специализациями `stringstream` и `wstringstream` для ввода данных из строк и вывода данных в строки

- Шаблонный класс `basic_stringbuf<>` со специализациями `stringbuf` и `wstringbuf`, используемыми другими строковыми потоковыми классами для ввода и вывода символов

Эти классы имеют такое же отношение к базовым потоковым классам, как и файловые потоковые классы. Эта иерархия классов изображена на рис. 15.3.

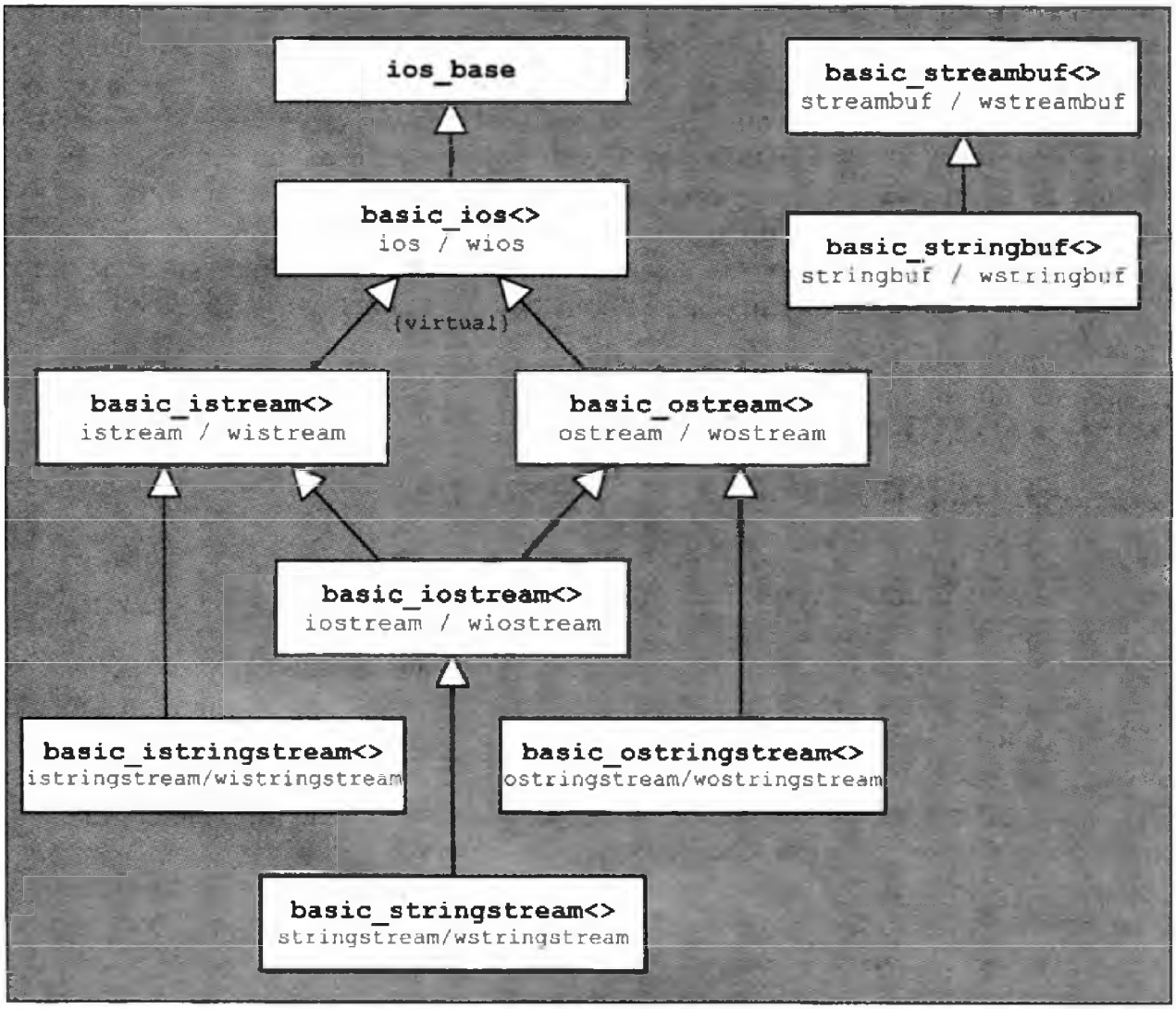


Рис. 15.3. Иерархия строковых потоковых классов

Классы объявлены в заголовочном файле `<sstream>` следующим образом:

```
namespace std {
   template <typename charT,
             typename traits = char_traits<charT>,
             typename Allocator = allocator<charT> >
             class basic_istringstream;
   typedef basic_istringstream<char> istringstream;
   typedef basic_istringstream<wchar_t> wistringstream;

   template <typename charT,
             typename traits = char_traits<charT>,
             typename Allocator = allocator<charT> >
```

```
            class basic_ostringstream;
   typedef basic_ostringstream<char> ostringstream;
   typedef basic_ostringstream<wchar_t> wostringstream;

   template <typename charT,
             typename traits = char_traits<charT>,
             typename Allocator = allocator<charT> >
             class basic_stringstream;
   typedef basic_stringstream<char> stringstream;
   typedef basic_stringstream<wchar_t> wstringstream;

   template <typename charT,
             typename traits = char_traits<charT>,
             typename Allocator = allocator<charT> >
             class basic_stringbuf;
   typedef basic_stringbuf<char> stringbuf;
   typedef basic_stringbuf<wchar_t> wstringbuf;
}
```

Основной функцией в интерфейсе строковых потоковых классов является функция-член `str()`, которая используется для управления буфером строкового потокового класса (табл. 15.38).

Таблица 15.38. Основные операции над строковыми потоками

Функция-член	Описание
`str()`	Возвращает буфер в виде строки
`str(`*string*`)`	Задает содержимое буфера с помощью аргумента *string*

Возможности строковых потоков демонстрирует следующая программа:

```
// io/sstream1.cpp

#include <iostream>
#include <sstream>
#include <bitset>
using namespace std;

int main()
{
   ostringstream os;

   // десятичное и шестнадцатеричное числа
   os << "dec: " << 15 << hex << " hex: " << 15 << endl;
   cout << os.str() << endl;

   // добавляем число с плавающей точкой и битовое множество
   bitset<15> b(5789);
   os << "float: " << 4.67 << " bitset: " << b << endl;

   // заменяем восьмеричным числом
   os.seekp(0);
   os << "oct: " << oct << 15;
```

```
   cout << os.str() << endl;
}
```

Программа выводит такой результат:

```
dec: 15 hex: f

oct: 17 hex: f
float: 4.67 bitset: 001011010011101
```

Сначала в поток os записывается десятичное и шестнадцатеричное числа. Затем добавляются число с плавающей точкой и битовое множество (записанное в двоичном виде). С помощью функции seekp() позиция записи перемещается в начало потока. Итак, следующий вызов оператора << записывает данные в начало потока, перезаписывая начало существующего строкового потока. Однако символы, которые не были перезаписаны, остаются корректными. Для того чтобы удалить текущее содержимое из потока, можно использовать функцию str(), записывающую в буфер новое содержимое:

```
strm.str("");
```

Первые строки, записанные в поток os, завершаются символом endl. Это значит, что строки завершаются символом перехода на новую строку. Поскольку строка выводится с манипулятором endl, в поток записываются подряд два символа перехода на новую строку. Это объясняет пустую строку при выводе.

До принятия стандарта C++11 программисты часто делали типичную ошибку: забывали извлекать строки из строкового потока с помощью функции str() и прямо записывали данные в поток. С точки зрения компилятора это возможно и разумно, потому что существовало неявное преобразование в тип void* (см. раздел 15.4.3). В результате состояние потока выводилось в виде адреса, а не значения (см. раздел 15.3.3). После принятия стандарта C++11 это преобразование было заменено явным преобразованием в тип bool, так что передача строкового потока оператору вывода << без вызова функции str() стало невозможным.

При записи данных в строковый поток вывода часто используются пользовательские операции вывода (см. раздел 15.11.1).

Строковые потоки ввода используются в основном для форматированного чтения данных из существующих строк. Например, часто проще прочитать данные строка за строкой, а затем анализировать каждую строку отдельно. Следующие операторы считывают целочисленную переменную x со значением 3 и переменную с плавающей точкой f со значением 0.7 из строки s:

```
int x;
float f;
std::string s = "3.7";

std::istringstream is(s);
is >> x >> f;
```

Строковый поток можно создать с помощью флагов, управляющих режимами открытия файлов (см. раздел 15.9.3) и/или существующей строки. Если установлен флаг ios::ate, то символы, записанные в строковый поток, можно добавлять к существующей строке[15]:

[15] В настоящее время не совсем понятно, имеет ли то же действие флаг ios::app, поэтому его использование противоречит требованию переносимости программ.

```
std::string s("value: ");
...
std::ostringstream os (s, std::ios::out|std::ios::ate);
os << 77 << " " << std::hex << 77 << std::endl;
std::cout << os.str();   // выводит: value: 77 4d
std::cout << s;          // выводит: value:
```

Легко видеть, что строка, возвращенная функцией `str()`, представляет собой копию строки `s`, к которой добавлено десятичное и шестнадцатеричное представление числа `77`. Сама строка `s` остается без изменений.

15.10.2. Семантика перемещения для строковых потоков

После принятия стандарта C++11 строковые потоки поддерживают rvalue-семантику и семантику перемещения. Потоки `ostream` обеспечивают операцию вывода, а потоки `istream` — операцию ввода, получающую rvalue-ссылку для потока. Это позволяет использовать временные объекты потоков. Например, можно вставить данные во временный строковый поток[16]:

```
// io/sstream2.cpp

#include <iostream>
#include <sstream>
#include <string>
#include <tuple>
#include <utility>
using namespace std;

tuple<string,string,string> parseName(string name)
{
   string s1, s2, s3;
   istringstream(name) >> s1 >> s2 >> s3;
   if (s3.empty()) {
       return tuple<string,string,string>(move(s1),"",move(s2));
   }
   else {
       return tuple<string,string,string>(move(s1),move(s2),move(s3));
   }
}

int main()
{
   auto t1 = parseName("Nicolai M. Josuttis");
   cout << "firstname: " << get<0>(t1) << endl;
   cout << "middle: " << get<1>(t1) << endl;
   cout << "lastname: " << get<2>(t1) << endl;
   auto t2 = parseName("Nico Josuttis");
   cout << "firstname: " << get<0>(t2) << endl;
   cout << "middle: " << get<1>(t2) << endl;
```

[16] Благодарю Даниэля Крюглера (Daniel Krügler) за этот пример.

```
    cout << "lastname: " << get<2>(t2) << endl;
}
```

До принятия стандарта C++11 необходимо было писать код

```
istringstream is(name);
is >> s1 >> s2 >> s3;
```

вместо

```
istringstream(name) >> s1 >> s2 >> s3;
```

Кроме того, строковые потоки теперь поддерживают семантику перемещения и обмена, имеют конструктор перемещения, операцию перемещающего присваивания и функцию-член `swap()`. Таким образом, строковый поток можно передавать функции как аргумент и возвращать из функции.

15.10.3. Потоковые классы char*

Потоковые классы `char*` поддерживаются только для обеспечения обратной совместимости. Их интерфейс уязвим для ошибок, а классы часто используются неправильно. Тем не менее они продолжают использоваться, и поэтому мы приведем их краткое описание. Отметим, что стандартная версия, описанная здесь, имеет немного измененный старый интерфейс.

В этом подразделе вместо термина *строка* используется термин *последовательность символов*, потому что последовательность символов, поддерживаемая потоковыми классами `char*`, не всегда заканчивается (завершающим) нулевым символом и не является настоящей строкой.

Потоковые классы `char*` определены только для символьного типа `char`. К ним относятся

- классы `istrstream` для чтения символьных последовательностей (строковый поток ввода);
- класс `ostrstream` для записи символьных последовательностей (строковый поток вывода);
- класс `strstream` для чтения и записи символьных символов;
- класс `strstreambuf`, использующийся как потоковый буфер для потоков `char*`.

Потоковые классы `char*` определены в заголовочном файле `<strstream>`.

Поток класса `istrstream` можно инициализировать символьной последовательностью (типа `char*`), которая завершается (завершающим) нулевым символом `0` или имеет длину, заданную аргументом. Как правило, эти потоки используются для чтения и записи полных строк:

```
char buffer[1000]; // буфер для по крайней мере 999 символов

// читаем строку
std::cin.get(buffer,sizeof(buffer));

// читаем/обрабатываем строку как поток
```

```
std::istrstream input(buffer);
...
input >> x;
```

Символ `char*` для записи может либо поддерживать символьную последовательность, которая при необходимости увеличивается, либо инициализироваться буфером фиксированного размера. Используя флаги `ios::app` или `ios:ate`, можно добавлять символы в символьную последовательность, которая уже хранится в буфере.

При использовании потока `char*` в качестве строки следует проявлять осторожность. В отличие от строковых потоков потоки `char*` не всегда управляют памятью, которая используется для хранения символьной последовательности.

Функция-член `str()` предоставляет вызывающей стороне доступ к символьной последовательности вместе с ответственностью за управление соответствующей памятью. Если поток данных не был инициализирован буфером фиксированного размера (за который поток никогда не отвечает), должны соблюдаться три правила.

1. Если поток данных не был инициализирован буфером фиксированного размера, то права владения памятью передаются вызывающей стороне и поэтому символьная последовательность должна освобождаться. Однако нет гарантии, как именно память была на самом деле выделена[17], вызывать `delete[]` не всегда безопасно. Лучше всего вернуть память потоку с помощью вызова функции `freeze()` с аргументом `false` (пример приводится ниже).
2. Вызов функции `str()` запрещает потоку дальнейшее изменение символьной последовательности. Функция неявно вызывает функцию `freeze()`, которая блокирует последовательность символов. Это делается для того, чтобы избежать трудностей при недостаточно большом буфере и при необходимости выделять новую память.
3. Функция-член `str()` не добавляет (завершающий) нулевой символ (`'\0'`). Этот символ должен отдельно присоединяться к потоку для завершения символьной последовательности. Для этого можно использовать манипулятор `ends`. Некоторые реализации присоединяют символ завершения строки автоматически, но это поведение нарушает переносимость программы.

Следующий пример демонстрирует использование потока `char*`:

```
float x;
...
// создаем и заполняем поток char*
// - не забывайте об ends или '\0' !!!
std::ostrstream buffer;    // динамический буфер потока
buffer << "float x: " << x << std::ends;
// передаем получившуюся С-строку в функцию foo() и возвращаем
// память потоку buffer
char* s = buffer.str();
foo(s);
buffer.freeze(false);
```

За счет дополнительных манипуляций можно восстановить нормальное состояние заблокированного потока `char*`. Для этого необходимо вызвать функцию-член `freeze()`

[17] Существует конструктор, получающий указатели на две функции: для выделения и освобождения памяти.

с аргументом false. С помощью этой операции права владения символьной последовательностью возвращаются потоковому объекту. Это единственный безопасный способ освободить память для символьной последовательности. Продемонстрируем это следующим примером:

```
float x;
...
std::ostrstream buffer; // динамический поток char*

// заполняем поток char*
buffer << "float x: " << x << std::ends;

// передаем получившуюся С-строку функции foo()
// - блокируем поток char*
foo(buffer.str());

// разблокируем поток char*
buffer.freeze(false);

// переносим позицию записи в начало
buffer.seekp (0, ios::beg);

// заново заполняем поток char*
buffer << "once more float x: " << x << std::ends;

// снова передаем полученную С-строку функции foo()
// - блокируем поток char*
foo(buffer.str());

// возвращаем память потоку buffer
buffer.freeze(false);
```

Проблемы, связанные с блокировкой буфера, в строковых потоковых классах не возникают в основном потому, что строки копируются и аккуратно используют память.

15.11. Операции ввода-вывода для пользовательских типов

Как упоминалось ранее, главным преимуществом потоков над старыми средствами ввода-вывода в языке С является возможность расширения потокового механизма на пользовательские типы. Для этого необходимо перегрузить операторы << и >>. В следующем подразделе рассматривается пример использования потоков данных для вывода правильных дробей.

15.11.1. Реализация операций вывода

В выражениях, содержащих оператор вывода <<, левый операнд определяет поток данных, а правый — объект, который записывается в этот поток.

```
поток << объект
```

В соответствии с правилами языка C++ это можно интерпретировать следующим образом.

1. Как *поток*.operator<<(*объект*)
2. Как operator<<(*поток*,*объект*)

Первый способ используется для встроенных типов. Для пользовательских типов необходимо применять второй способ, потому что потоковые классы закрыты для расширения. Все, что нужно сделать для этого, — реализовать глобальный оператор << для пользовательского типа. Это довольно просто, если не требуется получать доступ к закрытым членам объектов (этот вопрос будет рассмотрен позднее).

Например, чтобы вывести на печать объект класса Fraction в формате *числитель/знаменатель*, можно написать следующую функцию:

```
// io/fraclout.hpp

#include <iostream>
inline
std::ostream& operator << (std::ostream& strm, const Fraction& f)
{
   strm << f.numerator() << '/' << f.denominator();
   return strm;
}
```

Эта функция выводит числитель и знаменатель, разделенные символом '/', в поток данных, передаваемый в виде аргумента. Поток может быть файловым, строковым или каким-то еще. Для создания цепочек операций вывода и проверки состояния потока одновременно с выводом функция возвращает ссылку на поток.

У этой простой формы есть два основных недостатка.

1. Поскольку в сигнатуре функции используется класс ostream, ее можно применять только к потокам символов char. Если функция предназначается только для Западной Европы и Северной Америки, проблемы не возникают. С другой стороны, создать более универсальную версию несложно, поэтому следует по крайней мере рассмотреть такую возможность.
2. Другая проблема возникает при задании ширины поля. В данном случае результат окажется не тем, который можно было бы ожидать. Ширина поля будет относиться только к ближайшей операции вывода, в данном случае — к выводу числителя. Таким образом, операторы

   ```
   Fraction vat(19,100); // В Германии налог НДС равен 19%
   std::cout << "VAT: \"" << std::left << std::setw(8)
             << vat << '"' << std::endl;
   ```

   выводят следующий результат:

   ```
   VAT: "19      /100"
   ```

Следующая версия решает обе проблемы:

```
// io/frac2out.hpp

#include <iostream>
```

```
#include <sstream>

template <typename charT, typename traits>
inline
std::basic_ostream<charT,traits>&
operator << (std::basic_ostream<charT,traits>& strm,
             const Fraction& f)
{
   // строковый поток
   // - с тем же форматом
   // - без специальной ширины поля
   std::basic_ostringstream<charT,traits> s;
   s.copyfmt(strm);
   s.width(0);

   // заполняем строковый поток
   s << f.numerator() << '/' << f.denominator();

   // выводим строковый поток
   strm << s.str();
   return strm;
}
```

Оператор превратился в шаблонную функцию, параметризованную для всех разновидностей потоков. Проблема с шириной поля решается записью числителя в строковый поток без указания конкретной ширины. Созданная строка затем передается в поток, заданный аргументом. В результате символьное представление дроби выводится с помощью одной операции записи, к которой применяется ширина поля.

В результате операторы

```
Fraction vat(19,100); // В Германии ...
std::cout << "VAT: \"" << std::left << std::setw(8)
          << vat << '"' << std::endl;
```

выводят следующий результат:

```
VAT: "19/100  "
```

Пользовательская перегрузка оператора << для типов из пространства имен `std` имеет ограничения. Дело в том, что ее нельзя использовать в ситуациях, в которых используется *ADL-поиск* (*поиск, зависящий от аргумента*, или *поиск Кёнига*), например, когда используются итераторы потоков вывода.

```
template <typename T1, typename T2>
std::ostream& operator << (std::ostream& strm, const std::pair<T1,T2>& p)
{
   return strm << "[" << p.first << "," << p.second << "]";
}

std::pair<int,long> p(42,77777);
std::cout << p << std::endl; // OK

std::vector<std::pair<int,long>> v;
```

```
...
std::copy(v.begin(),v.end(), // ОШИБКА: не компилируется:
          std::ostream_iterator<std::pair<int,long>>(std::cout,"\n"));
```

15.11.2. Реализация операций ввода

Операции ввода реализуются в соответствии с теми же принципами, что и операции вывода. Однако при вводе могут возникнуть проблемы чтения. Как правило, функции чтения должны особым образом обрабатывать неудачный ввод.

При реализации функции чтения можно выбирать между простотой и гибкостью. Например, в следующей функции используется простой подход: дробь считывается без проверки возможных ошибок:

```
// io/fraclin.hpp

#include <iostream>

inline
std::istream& operator >> (std::istream& strm, Fraction& f)
{
   int n, d;

   strm >> n;           // читаем значение числителя

   strm.ignore();       // пропускаем '/'
   strm >> d;           // читаем значение знаменателя

   f = Fraction(n,d);   // присваиваем всю дробь

   return strm;
}
```

Проблема заключается в том, что такая реализация подходит только для потоков данных с символьным типом `char`. Кроме того, она не проверяет, действительно ли два числа разделяются символом `'/'`.

Другая проблема возникает при вводе неопределенных значений. Если знаменатель считываемой дроби равен нулю, то значение дроби становится неопределенным. Проблема обнаруживается в конструкторе класса `Fraction`, который вызывается выражением `Fraction(n,d)`. Однако это означает, что ошибки форматирования автоматически обрабатываются в классе `Fraction`. Поскольку на практике ошибки форматирования обычно регистрируются на уровне потоков данных, лучше в этом случае установить флаг `ios_base::failbit`.

В заключение дробь, переданную по ссылке, можно модифицировать даже при неудачном вводе. Это может произойти, например, когда числитель вводится успешно, а при чтении знаменателя возникает сбой. Такое поведение противоречит общепринятым правилам, установленным стандартными операциями ввода, и его лучше всего избегать. Операция чтения должна либо завершаться успешно, либо не иметь последствий.

Рассмотрим улучшенную реализацию программы, в которой указанные проблемы не возникают. Кроме того, эта программа более гибкая, поскольку параметризация позволяет примерять ее для любых типов потоков данных:

```
// io/frac2in.hpp

#include <iostream>

template <typename charT, typename traits>
inline
std::basic_istream<charT,traits>&
operator >> (std::basic_istream<charT,traits>& strm, Fraction& f)
{
   int n, d;

   // считываем числитель
   strm >> n;

   // если числитель прочитан успешно,
   // - считываем '/' и знаменатель
   if (strm.peek() == '/') {
       strm.ignore();
       strm >> d;
   }
   else {
       d = 1;
   }

   // если знаменатель равен нулю
   // - устанавливаем флаг failbit как признак ошибки ввода-вывода
   if (d == 0) {
       strm.setstate(std::ios::failbit);
       return strm;
   }

   // если все хорошо,
   // - изменяем значение дроби
   if (strm) {
       f = Fraction(n,d);
   }

   return strm;
}
```

На этот раз знаменатель вводится только в том случае, если за первым числом следует символ `'/'`, в противном случае по умолчанию считается, что знаменатель равен 1, а целое число интерпретируется как дробь. Таким образом, для целых чисел знаменатель не обязателен.

Эта реализация также проверяет, не равен ли прочитанный знаменатель нулю. В этом случае устанавливается флаг `ios_base::failbit`, что может сопровождаться генерацией соответствующего исключения (раздел 15.4.4). Если знаменатель равен нулю, обработка может быть другой. Например, исключение можно сгенерировать непосредственно или не проверять знаменатель, чтобы исключение генерировалось классом `Fraction`.

В заключение проверяется состояние потока данных, и новое значение присваивается дроби, только если ввод был выполнен без ошибок. Последнюю проверку следует проводить всегда, чтобы значение объекта происходило, только если чтение прошло успешно.

Разумеется, целесообразность чтения целых чисел в виде дроби можно поставить под сомнение. Кроме того, можно было бы исправить еще несколько моментов: например, символ '/' должен следовать за числителем без разделяющих пробелов. Однако знаменателю может предшествовать произвольное количество пробелов, которые обычно игнорируются. Это свидетельствует о сложностях, связанных с чтением нетривиальных структур данных.

15.11.3. Ввод и вывод с помощью вспомогательных функций

Если реализация операции ввода-вывода требует доступа к закрытым данным объекта, то стандартные операции должны делегировать фактическую работу вспомогательным функциям-членам классов. Этот подход позволяет создавать полиморфные функции чтения и записи. Рассмотрим пример:

```
class Fraction {
   ...
   public:
      virtual void printOn (std::ostream& strm) const;  // вывод
      virtual void scanFrom (std::istream& strm);       // ввод
      ...
};

std::ostream& operator << (std::ostream& strm, const Fraction& f)
{
   f.printOn (strm);
   return strm;
}

std::istream& operator >> (std::istream& strm, Fraction& f)
{
   f.scanFrom (strm);
   return strm;
}
```

Типичным примером является прямой доступ к числителю и знаменателю дроби во время ввода.

```
void Fraction::scanFrom (std::istream& strm)
{
   ...
   // присваиваем значения непосредственно компонентам
   num = n;
   denom = d;
}
```

Если класс не будет использоваться в качестве базового, то операции ввода-вывода можно объявить дружественными для этого класса. Однако такой подход значительно ограничивает возможности при наследовании. Дружественные функции не могут быть виртуальными, поэтому могут быть вызваны неправильные функции. Например, если в аргументе операции ввода передается ссылка на базовый класс, которая на самом деле ссылается на объект производного класса, то для нее будет вызвана операция из базового

класса. Для того чтобы избежать этой проблемы, производные классы не должны реализовывать собственные операции ввода-вывода. Таким образом, описанная выше реализация является более универсальной, чем реализация на основе дружественных функций. Она должна рассматриваться как стандартный подход, хотя в большинстве примеров применяются дружественные функции.

15.11.4. Пользовательские флаги форматов

При создании пользовательских операций ввода-вывода часто желательно иметь специальные флаги форматирования, соответствующие этим операциям и устанавливаемые соответствующим манипулятором. Например, было бы хорошо, если бы операция вывода для дроби, показанная выше, конфигурировалась так, чтобы символы '/', разделяющие числители и знаменатели, были окружены пробелами.

Эту возможность предоставляют объекты потоков — в них предусмотрен механизм связывания данных с потоком. Он позволяет задать нужные значения, например, с помощью манипулятора, и прочитать их позднее. В классе `ios_base` определены две функции, `iword()` и `pword()`, которые при вызове получают индекс типа `int` и возвращают по нему соответствующее значение `long&` или `vold*&`. Предполагается, что функции `iword()` и `pword()` обращаются к объектам типа `long` или `void*` в массиве произвольного размера, хранящемся в объекте потока. Флаги форматирования, сохраняемые для потока, располагаются по одному и тому же индексу для всех потоков. Статическая функция-член `xalloc()` класса `ios_base` используется для получения индекса, который еще не применялся для этой цели.

В исходном состоянии объекты, доступ к которым осуществляется функциями `iword()` и `pword()`, равны 0. Это значение может использоваться для форматирования по умолчанию или как признак того, что к данным еще не обращались. Рассмотрим пример:

```
// получаем индекс для новых данных в потоке ostream
static const int iword_index = std::ios_base::xalloc();

// определяем манипулятор для установки этих данных
std::ostream& fraction_spaces (std::ostream& strm)
{
   strm.iword(iword_index) = true;
   return strm;
}

std::ostream& operator<< (std::ostream& strm, const Fraction& f)
{
   // запрос к данным из потока ostream
   // - если условие выполняется,
   //   вставлять пробелы между числителем и знаменателем
   // - если условие не выполняется,
   //   вставлять пробелы между числителем и знаменателем
   if (strm.iword(iword_index)) {
       strm << f.numerator() << " / " << f.denominator();
   }
   else {
       strm << f.numerator() << "/" << f.denominator();
   }
   return strm;
}
```

В этом примере используется простой подход к реализации операции вывода, потому что его основная цель — демонстрация функции `iword()`. Флаг форматирования считается булевым значением, определяющим, следует ли между числителем и знаменателем вставлять пробел. В первой строке функция `ios_base::xalloc()` возвращает индекс, который может использоваться для хранения флага форматирования. Результат вызова сохраняется в константе, поскольку это значение никогда не изменяется. Функция `fraction_spaces()` — это манипулятор присваивания значения `true` переменной типа `int`, хранящейся по индексу `iword_index` в целочисленном массиве, связанном с потоком `strm`. Операция вывода извлекает это значение и выводит дробь в соответствии с состоянием флага. Если флаг равен `false`, то по умолчанию дробь выводится без пробелов, а если `true` — символ `'/'` окружается двумя пробелами.

Функции `iword()` и `pword()` возвращают ссылки на объекты типа `long` или `void*`. Эти ссылки остаются корректными только до следующего вызова функции `iword()` или `pword()` с соответствующим объектом потока данных или до уничтожения объекта потока. Обычно результаты функций `iword()` и `pword()` сохраняться не должны[18]. Предполагается, что доступ происходит достаточно быстро, хотя хранение данных в массиве не обязательно.

Функция `copyfmt()` копирует всю информацию о формате (см. раздел 15.7.1), в том числе и массивы, с которыми работают функции `iword()` и `pword()`. Это может породить проблемы для объектов, сохраняемых в потоке с помощью функции `pword()`. Например, если значение — это адрес объекта, то вместо объекта будет скопирован только адрес. При копировании адресов изменение формата в одном потоке может распространяться на другие потоки. Кроме того, может быть желательным, чтобы объект, ассоциированный с потоком функцией `pword()`, удалялся при уничтожении потока. Следовательно, для таких объектов желательно реализовать глубокое, а не поверхностное копирование.

Для реализации глубокого копирования при необходимости и для удаления объекта при уничтожении потока в классе `ios_base` определен механизм обратного вызова. Функция `register_callback()` регистрирует функцию, вызываемую при выполнении определенных условий для объекта класса `ios_base`. Эта функция определяется следующим образом:

```
namespace std {
   class ios_base {
      public:
         // виды событий, служащих причиной обратного вызова
         enum event { erase_event, imbue_event, copyfmt_event };
         // типы обратных вызовов
         typedef void (*event_callback) (event e, ios_base& strm,
                                         int arg);
         // функция регистрации обратных вызовов
         void register_callback (event_callback cb, int arg);
         ...
   };
}
```

Функция `register_callback()` получает два аргумента: указатель на функцию и переменную типа `int`. При вызове зарегистрированной функции в третьем аргументе

[18] В принципе возвращаемые указатели и ссылки не должны сохраняться, потому что срок жизни объекта типа `long`, на которые они ссылаются, точно не известен.

передается переменная типа int. Например, она может использоваться для идентификации индекса для функции pword(), т.е. задавать элемент массива, который должен быть обработан. Аргумент strm, передаваемый функции обратного вызова, содержит объект класса ios_base, вызвавший эту функцию. Аргумент e идентифицирует причину вызова. Эти причины перечислены в табл. 15.39.

Таблица 15.39. Причины обратных вызовов

Событие	Причина
ios_base::imbue_event	Функция imbue() установила локальный контекст
ios_base::erase_event	Уничтожен поток или вызвана функция copyfmt()
ios_base::copy_event	Вызвана функция copyfmt()

При вызове функции copyfmt() обратные вызовы, зарегистрированные для объекта, вызвавшего функцию copyfmt(), выполняются дважды. Еще до начала копирования они вызываются с аргументом erase_event и выполняют предварительную очистку (например, удаляют объекты, хранящиеся в массиве pword()). Выполняются обратные вызовы, зарегистрированные для данного объекта. После копирования флагов форматирования функции, включая список функций обратного вызова из потокового аргумента, обратного вызова вызываются снова с аргументом copy_event. Этот проход может использоваться, например, для организации глубокого копирования объектов, хранящихся в массиве pword(). Отметим, что функции обратного вызова тоже копируются, а исходный список зарегистрированных функций удаляется. Следовательно, при втором проходе будут вызваны только что скопированные функции.

Механизм обратного вызова очень примитивен. Он не позволяет отменять регистрацию функций обратного вызова, за исключением вызова функции copyfmt() с аргументом, не имеющим зарегистрированных функций. Кроме того, повторная регистрация функции обратного вызова даже с тем же аргументом приведет к повторному обратному вызову. Тем не менее библиотека гарантирует, что функции будут вызваны в порядке, обратном порядку их регистрации. Таким образом, функция обратного вызова, зарегистрированная из другой функции, не вызывается до следующего включения механизма обратного вызова.

15.11.5. Соглашения создания пользовательских операций ввода-вывода

Ниже перечислены некоторые соглашения, которые должны соблюдаться при реализации пользовательских операторов ввода-вывода. Эти соглашения соответствуют типичному поведению стандартных операторов.

- Формат вывода должен допускать определение оператора ввода, читающего данные без потери информации. Для строк эта задача практически невыполнима из-за проблем с пробелами. Пробел внутри строки невозможно отличить от пробела, разделяющего две строки.
- При вводе-выводе должна учитываться текущая спецификация формата потока. Прежде всего это относится к ширине поля при выводе. Особенно это относится к ширине записи.
- При возникновении ошибок должен быть установлен соответствующий флаг состояния.

- Ошибки не должны изменять состояние объекта. Если операция читает несколько объектов данных, промежуточные результаты сохраняются во вспомогательных объектах до окончательного принятия значения.
- Вывод не должен завершаться символом перехода на новую строку, в основном из-за того, что это не позволяет выводить другие объекты в той же строке.
- Даже слишком большие данные должны считываться полностью. После чтения необходимо установить соответствующий флаг ошибки, а возвращаемое значение должно содержать полезную информацию, например, максимальное значение.
- При обнаружении ошибки форматирования по возможности никакие символы не должны быть считаны.

15.12. Связывание потоков ввода и вывода

Иногда возникает необходимость связать два потока данных. Например, может потребоваться, чтобы перед вводом текста на экран выводилось предложение ввести данные. Другой пример — чтение и запись в одном и том же потоке. В основном это относится к файлам. Третий пример — необходимость манипулировать одним и тем же потоком в разных форматах. Все эти примеры рассматриваются в данном разделе.

15.12.1. Нежесткое связывание с помощью функции tie()

Поток можно связать с потоком вывода. Это значит, что буферы обоих потоков синхронизируются, т.е. перед каждой операцией ввода или вывода в другой поток буфер потока вывода очищается. Иначе говоря, для потоков вывода вызывается функция `flush()`. В табл. 15.40 перечислены функции-члены, определенные в классе `basic_ios`, для связывания двух потоков.

Таблица 15.40. Связывание двух потоков

Функция-член	Описание
`tie()`	Возвращает указатель на поток вывода, связанный с указанным потоком
`tie(`*`ostream*`* *`strm`*`)`	Связывает поток вывода, заданный аргументом, с потоком и возвращает указатель на предыдущий поток вывода, который был связан с потоком, если он существовал

Функция `tie()`, вызванная без аргументов, возвращает указатель на текущий поток вывода, связанный с данным потоком. Для того чтобы связать поток с новым потоком вывода, необходимо передать указатель на этот поток вывода с помощью аргумента функции `tie()`. Аргумент определен как указатель, поскольку с его помощью можно также передавать `nullptr` (или `0`, или `NULL`). Этот аргумент означает "разрыв" связи между потоками. Если потока вывода, связанного с данным потоком, не существует, функция возвращает значение `nullptr` или `0`. Каждый поток данных может быть связан только с одним потоком вывода, но поток вывода может быть связан с несколькими потоками данных.

По умолчанию стандартный поток ввода связан со стандартным потоком вывода с помощью механизма

```
// стандартные связи:

std::cin.tie (&std::cout);
std::wcin.tie (&std::wcout);
```

Это гарантирует, что сообщение, приглашающее ввести данные, будет очищено до запроса на ввод. Например, в операторах

```
std::cout << "Please enter x: ";
std::cin >> x;
```

функция `flush()` вызывается неявно для потока `cout` перед вводом переменной `x`.

Для разрыва связи между двумя потоками функции `tie()` следует передать аргумент со значением `nullptr` (или `0`, или `NULL`). Рассмотрим пример:

```
// разрыв связи между потоком cin и любым потоком вывода
std::cin.tie (nullptr);
```

Это может повысить производительность программы за счет предотвращения необязательных дополнительных очисток потоков (производительность потоков обсуждается в разделе 15.14.2).

Поток вывода можно связать с другим потоком вывода. Например, в следующих операторах обычный поток вывода очищается до записи сообщений в поток ошибок:

```
// связывание потоков cout и cerr
std::cerr.tie (&std::cout);
```

15.12.2. Жесткое связывание с помощью потоковых буферов

Функция `rdbuf()` осуществляет жесткое связывание потоков с помощью общего потокового буфера (табл. 15.41). Ее предназначение рассматривается в следующих подразделах.

Таблица 15.41. Доступ к потоковому буферу

Функция-член	Описание
`rdbuf()`	Возвращает указатель на потоковый буфер
`rdbuf(`*streambuf**`)`	Инсталлирует потоковый буфер, на который ссылается аргумент, и возвращает потоковый буфер, использовавшийся ранее

Функция-член `rdbuf()` позволяет нескольким потоковым объектам читать из одного канала ввода или записывать в один канал вывода без нарушения порядка ввода-вывода. Однако использование нескольких потоковых буферов создает проблемы из-за буферизации операций ввода-вывода. Таким образом, если разные каналы используют разные буфера одного и того же канала ввода-вывода, данные из одного канала могут проходить по другому. Для инициализации потока с помощью потокового буфера, передаваемого как аргумент, используется дополнительный конструктор классов `basic_istream` и `basic_ostream`. Например:

```
// io/streambuffer1.cpp

#include <iostream>
```

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

int main()
{
   // поток для шестнадцатеричного стандартного вывода
   ostream hexout(cout.rdbuf());
   hexout.setf (ios::hex, ios::basefield);
   hexout.setf (ios::showbase);

   // переключаемся между десятичным и шестнадцатеричным выводами
   hexout << "hexout: " << 177 << " ";
   cout << "cout: " << 177 << " ";
   hexout << "hexout: " << -49 << " ";
   cout << "cout: " << -49 << " ";
   hexout << endl;
}
```

Отметим, что деструктор классов `basic_istream` и `basic_ostream` *не удаляет* соответствующий потоковый буфер (в любом случае эти классы его не открывают). Следовательно, чтобы передать поток при вызове, вместо ссылки на поток можно передать указатель на потоковый буфер.

```
// io/streambuffer2.cpp

#include <iostream>
#include <fstream>

void hexMultiplicationTable (std::streambuf* buffer, int num)
{
   std::ostream hexout(buffer);
   hexout << std::hex << std::showbase;

   for (int i=1; i<=num; ++i) {
        for (int j=1; j<=10; ++j) {
             hexout << i*j << ' ';
        }
        hexout << std::endl;
   }
} // НЕ закрывает буфер

int main()
{
   using namespace std;
   int num = 5;

   cout << "We print " << num
        << " lines hexadecimal" << endl;

   hexMultiplicationTable(cout.rdbuf(),num);

   cout << "That was the output of " << num
        << " hexadecimal lines " << endl;
}
```

Преимущество такого подхода заключается в том, что после модификации формат не нужно возвращать в исходное состояние, поскольку он относится к объекту потока, а не к буферу. Результат работы выглядит следующим образом:

```
We print 5 lines hexadecimal
0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa
0x2 0x4 0x6 0x8 0xa 0xc 0xe 0x10 0x12 0x14
0x3 0x6 0x9 0xc 0xf 0x12 0x15 0x18 0x1b 0x1e
0x4 0x8 0xc 0x10 0x14 0x18 0x1c 0x20 0x24 0x28
0x5 0xa 0xf 0x14 0x19 0x1e 0x23 0x28 0x2d 0x32
That was the output of 5 hexadecimal lines
```

Впрочем, недостаток этого подхода состоит в том, что создание и уничтожение объекта потока обходится дороже, чем простая установка и восстановление форматных флагов. Кроме того, уничтожение объекта потока не приводит к очистке буфера. Очищать буфер вывода необходимо "вручную".

Тот факт, что потоковый буфер не уничтожается, относится не только к классам `basic_istream` и `basic_ostream`. Другие потоковые классы уничтожают буфера, созданные ими же, но оставляют буфера, назначенные функцией `rdbuf()`.

15.12.3. Перенаправление стандартных потоков

В старых реализациях библиотеки IOStream глобальные потоки `cin`, `cout`, `cerr` и `clog` были объектами классов `istream_withassign` и `ostream_withassign`. Благодаря этому существовала возможность перенаправлять потоки, присваивая одни потоки другим. В настоящее время этот механизм был исключен из стандартной библиотеки C++. Тем не менее возможность перенаправления потоков была сохранена и расширена так, что теперь она может применяться ко всем потокам. Поток может быть перенаправлен с помощью назначения потокового буфера.

Назначение потоковых буферов представляет собой перенаправление потоков под управлением программы без участия операционной системы. Например, следующий фрагмент программы передает данные, отправленные в поток `cout`, не в стандартный канал вывода, а в файл `cout.txt`.

```
std::ofstream file ("cout.txt");
std::cout.rdbuf (file.rdbuf());
```

Для передачи всей информации о формате от одного потока другому можно использовать функцию `copyfmt()`.

```
std::ofstream file ("cout.txt");
file.copyfmt (std::cout);
std::cout.rdbuf (file.rdbuf());
```

Внимание! Объект `file` является локальным и уничтожается в конце блока. Это приводит к уничтожению соответствующего потокового буфера. Этим файловые потоки данных отличаются от "обычных", поскольку они создают свои объекты потоковых буферов во время конструирования и уничтожают их при уничтожении. Следовательно, в приведенном примере объект `cout` больше невозможно использовать для записи. Более того, его даже нельзя безопасно уничтожить по завершении программы. По этой причине прежний буфер следует *всегда* сохранять с последующим восстановлением! В следующем примере это делается в функции `redirect()`:

```
// io/streamredirect1.cpp

#include <iostream>
#include <fstream>
#include <memory>
using namespace std;

void redirect(ostream&);
int main()
{
   cout << "the first row" << endl;

   redirect(cout);

   cout << "the last row" << endl;
}

void redirect (ostream& strm)
{
   // сохраняем буфер потока вывода
   // - используем уникальный указатель с деструктором,
   //   гарантирующим восстановление буфера вывода в конце функции
   auto del = [&](streambuf* p) {
                 strm.rdbuf(p);
              };
   unique_ptr<streambuf,decltype(del)> origBuffer(strm.rdbuf(),del);

   // перенаправляем вывод в файл redirect.txt
   ofstream file("redirect.txt");
   strm.rdbuf (file.rdbuf());

   file << "one row for the file" << endl;
   strm << "one row for the stream" << endl;
} // автоматически закрывает файл И его буфер
```

Уникальный указатель (см. раздел 5.2.5) гарантирует, что даже при выходе из функции `resize()` из-за исключения исходный буфер вывода, хранящийся в объекте `origBuffer`, будет восстановлен[19].

Программа выводит следующий результат:

```
the first row
the last row
```

Файл `redirect.txt` содержит следующие строки:

```
one row for the file
one row for the stream
```

Как видим, данные, записанные в поток `cout` в функции `redirect()`, были переданы в файл, имя которого задано параметром `strm`. Данные, записанные в функции `main()`, после выполнения функции `redirect()` отправлены в восстановленный канал вывода.

[19] Благодарю Даниэля Крюглера за это замечание.

15.12.4. Потоки для чтения и записи

В последнем примере один и тот же поток используется как для чтения, так и для записи. Обычно файл открывается для чтения и записи с помощью класса `fstream`.

```
std::fstream file ("example.txt", std::ios::in | std::ios::out);
```

Кроме того, можно использовать два разных потоковых объекта — один для чтения, а второй для записи. Соответствующие объявления могут выглядеть примерно следующим образом:

```
std::ofstream out ("example.txt", ios::in | ios::out);
std::istream in (out.rdbuf());
```

Объявление потока `out` открывает файл. Объявление потока `in` использует буфер потока `out` для чтения из него. Отметим, что поток `out` должен открываться для чтения и записи. Если открыть его только для записи, то чтение из потока приведет к непредсказуемым последствиям. Кроме того, подчеркнем, что поток `in` определяется с типом `istream`, а не `ifstream`. Файл уже открыт, и у него есть соответствующий потоковый буфер. Необходим лишь второй потоковый объект. Как и в предыдущих примерах, файл закрывается при уничтожении объекта файлового потока `out`.

Можно также создать буфер файлового потока и указать его для двух потоковых объектов. Код выглядит примерно так:

```
std::filebuf buffer;
std::ostream out (&buffer);
std::istream in (&buffer);
buffer.open("example.txt", std::ios::in | std::ios::out);
```

Класс `filebuf` — это обычная специализация класса `basic_filebuf<>` для символьного типа `char`. Этот класс определяет класс потокового буфера, использующийся файловыми потоками.

Следующая программа представляет собой законченный пример. В цикле в файл записываются четыре строки. После каждой записи строки содержимое файла выводится в стандартный поток вывода.

```
// io/streamreadwrite1.cpp

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

int main()
{
   // открываем файл "example.dat" для чтения и записи
   filebuf buffer;
   ostream output(&buffer);
   istream input(&buffer);
   buffer.open ("example.dat", ios::in | ios::out | ios::trunc);

   for (int i=1; i<=4; i++) {
        // записываем одну строку
        output << i << ". line" << endl;

        // выводим все содержимое файла
        input.seekg(0);          // переходим в начало
```

```
        char c;
        while (input.get(c)) {
               cout.put(c);
         }
         cout << endl;
         input.clear();          // сбрасываем флаги eofbit и failbit
   }
}
```

Программа выводит следующие строки:

```
1. line

1. line
2. line

1. line
2. line
3. line

1. line
2. line
3. line
4. line
```

Несмотря на то что для чтения и записи используются два разных объекта потоков, позиции чтения и записи тесно связаны между собой. Функции `seekg()` и `seekp()` вызывают одну и ту же функцию-член потокового буфера[20]. Следовательно, для того чтобы вывести все содержимое файла, необходимо всегда устанавливать позицию чтения в начало файла. После вывода всего содержимого файла позиция чтения и записи снова перемещается в конец файла для присоединения новых строк.

Операции чтения и записи в одном и том же файле обязательно должны разделяться операцией позиционирования, за исключением выхода за конец файла во время чтения. Отсутствие операции позиционирования приведет к искажению содержимого файла или еще более фатальным ошибкам.

Как упоминалось ранее, вместо последовательной обработки символов все содержимое файла можно вывести одной командой, передав оператору << указатель на потоковый буфер (см. раздел 15.14.3).

```
std::cout << input.rdbuf();
```

15.13. Классы потоковых буферов

Как указывалось в разделе 15.2.1, потоки не выполняют операции чтения и записи непосредственно, а делегируют их потоковым буферам.

Общий интерфейс для работы с потоковыми буферами несложен (см. раздел 15.12.2).

- Функция `rdbuf()` возвращает указатель на потоковый буфер.
- Конструктор потока и его функция-член `rdbuf()` позволяют настраивать потоковый буфер в момент создания объекта или изменять потоковый буфер во время

[20] Эта функция различает, какие позиции должны модифицироваться: для чтения, записи, для чтения и записи. Одну позицию для чтения и записи поддерживают только стандартные потоковые буфера.

существования потока. В обоих случаях необходимо передавать указатель на потоковый буфер, который возвращается функцией `rdbuf()`.

Благодаря этому потоки могут записывать данные в одно и то же устройство вывода (см. раздел 15.12.2), перенаправлять потоки (см. раздел 15.12.3), выполнять операции чтения и записи с одним и тем же буфером (см. раздел 15.12.4), а также использовать для ввода и вывода другие кодировки символов, такие как UTF-8 или UTF-16/UCS-2 (см. раздел 16.4.4).

В этом разделе описывается работа классов потоковых буферов. В результате читатели не только получат более глубокое понимание процессов, протекающих при вводе и выводе, но и научатся определять новые каналы ввода-вывода. Прежде чем углубляться в детали операций над потоковыми буферами, сначала рассмотрим открытый интерфейс потоковых буферов.

15.13.1. Интерфейсы потоковых буферов

С точки зрения пользователя потокового буфера класс `basic_streambuf<>` представляет собой не более чем сущность, принимающую и отправляющую символы. Открытые функции для записи символов приведены в табл. 15.42.

Таблица 15.42. Открытые функции-члены для записи символов

Функция-член	Описание
`sputc(`*c*`)`	Отправляет символ *c* в потоковый буфер
`sputn(`*s*`, `*n*`)`	Отправляет *n* символов из последовательности *s* в потоковый буфер

Функция `sputc()` возвращает значение `traits_type::eof()` в случае ошибки, где `traits_type` — определение типа в классе `basic_streambuf`. Функция `sputn()` записывает количество символов, заданное вторым аргументом, пока потоковый буфер может их принимать. Эта функция никак не выделяет (завершающие) нулевые символы. Функция возвращает количество записанных символов.

Интерфейс для чтения символов из потокового буфера немного сложнее (табл. 15.43), потому что при вводе нужна возможность просмотра символа без его извлечения из буфера. Кроме того, желательно иметь возможность возвращать символы в потоковый буфер при лексическом анализе. Для этого в классах потоковых буферов предусмотрены соответствующие функции.

Таблица 15.43. Открытые функции-члены для чтения символов

Функция-член	Описание
`in_avail()`	Возвращает нижнюю границу доступных символов
`sgetc()`	Возвращает текущий символ без его извлечения из буфера
`sbumpc()`	Возвращает текущий символ с извлечением из буфера
`snextc()`	Извлекает текущий символ из буфера и возвращает следующий символ
`sgetn(`*b*`, `*n*`)`	Считывает *n* символов и сохраняет их в буфере *b*
`sputbackc(`*c*`)`	Возвращает символ *c* в потоковый буфер
`sungetc()`	Перемещается на шаг назад к предыдущему символу

Функция in_avail() проверяет минимальное количество доступных символов. Например, с ее помощью можно убедиться в том, что чтение не будет заблокировано при вводе с клавиатуры. С другой стороны, количество доступных символов может быть больше этого значения.

Пока потоковый буфер не достигнет конца потока, один из символов считается текущим. Функция sgetc() используется для получения текущего символа без перемещения к следующему символу. Функция sbumpc() читает текущий символ и перемещает указатель потока на следующий символ, который становится текущим. Последняя из функций чтения отдельных символов, snextc(), переходит к следующему символу и читает новый текущий символ. В качестве индикатора ошибки все три функции возвращают значение traits_type::eof(). Функция sgetn() читает из буфера последовательность символов и возвращает количество прочитанных символов. Максимальное количество считываемых символов передается в аргументе.

Функции sputbackc() и sungetc() возвращаются назад на одну позицию в потоке. В результате текущим становится предыдущий символ. Функция sputbackc() может использоваться для замены предыдущего символа другим символом. При вызове этих функций необходимо соблюдать осторожность. Часто вернуть можно только один символ.

И наконец, существует множество функций для доступа к объекту локального контекста, изменения позиции и управления буферизацией (табл. 15.44).

Таблица 15.44. Открытые функции для работы с потоковым буфером

Функция-член	Описание
pubimbue(*loc*)	Ассоциирует потоковый буфер с локальным контекстом *loc*
getloc()	Возвращает текущий локальный контекст
pubseekpos(*pos*)	Перемещает указатель текущей позиции потока в позицию, заданную абсолютным значением
pubseekpos(*pos*, *which*)	То же самое с указанием направления ввода-вывода
pubseekoff(*offset*, *rpos*)	Перемещает указатель текущей позиции потока в позицию, заданную относительным смещением
pubseekoff(*offset*, *rpos*, *which*)	То же самое с указанием направления ввода-вывода
pubsetbuf(*buf*, *n*)	Управление буферизацией

Функции pubimbue() и getloc() используются при интернационализации (см. раздел 15.8). Функция pubimbue() подключает новый объект локального контекста к потоковому буферу и возвращает ранее установленный объект локального контекста. Функция getloc() возвращает текущий объект локального контекста.

Функция pubsetbuf() предоставляет возможности управлять стратегией буферизации потоковых буферов. Тем не менее ее возможности зависят от конкретного класса потокового буфера. Например, вызов функции pubsetbuf() для буферов строковых потоков не имеет смысла. Даже для буферов файловых потоков данных применение этой функции является переносимым лишь в том случае, если она вызывается перед выполнением первой операции ввода-вывода или в виде pubsetbuf(nullptr,0), т.е. буфер не используется. Функция возвращает nullptr в случае неудачи и объект потокового буфера при успешном выполнении.

Функции pubseekoff() и pubseekpos() используются для управления текущей позицией чтения и/или записи. Эта позиция зависит от последнего аргумента, который имеет

тип ios_base::openmode и по умолчанию равен ios_base::in | ios_base::out. Если установлен флаг ios_base::in, то изменяется позиция чтения. Если установлен флаг ios_base::out, то изменяется позиция записи. Функция pubseekpos() перемещает указатель текущей позиции потока в абсолютную позицию, заданную первым аргументом, а функция pubseekoff() использует смещение, заданное по отношению к другой позиции. Смещение передается в первом аргументе. Позиция, по отношению к которой задается смещение, передается во втором аргументе и может быть равна ios_base::cur, ios_base::beg или ios_base::end (см. раздел 15.9.4). Обе функции возвращают новую текущую позицию или признак некорректной позиции. Для того чтобы обнаружить некорректную позицию, следует сравнить результат с объектом pos_type(off_type(-1)) (типы pos_type и off_type используются для определения позиций в потоках данных; см. раздел 15.9.4). Текущая позиция потока возвращается функцией pubseekoff().

```
sbuf.pubseekoff(0, std::ios::cur)
```

15.13.2. Итераторы потоковых буферов

Альтернативный способ неформатированного ввода-вывода использует классы итераторов потоковых буферов. Эти классы предоставляют корректные итераторы ввода и вывода, предназначенные для чтения или записи отдельных символов в потоковых буферах. Они совместимы с алгоритмами ввода-вывода алгоритмов из стандартной библиотеки C++.

Шаблонные классы istreambuf_iterator<> и ostreambuf_iterator<> используются для чтения или записи отдельных символов с объектами типа basic_streambuf<> соответственно. Эти классы определены в заголовочном файле <iterator>.

```
namespace std {
   template <typename charT,
             typename traits = char_traits<charT> >
             class istreambuf_iterator;
   template <typename charT,
             typename traits = char_traits<charT> >
             class ostreambuf_iterator;
}
```

Эти итераторы представляют собой особые разновидности потоковых итераторов, описанных в разделе 9.4.3. Единственное отличие заключается в том, что их элементами являются символы.

Итераторы потоковых буферов вывода

Покажем, как можно записать строку в потоковый буфер с помощью итератора ostreambuf_iterator:

```
// создаем итератор для буфера потока вывода cout
std::ostreambuf_iterator<char> bufWriter(std::cout);

std::string hello("hello, world\n");
std::copy(hello.begin(), hello.end(),  // источник: string
          bufWriter);                  // получатель: буфер вывода потока cout
```

В первой строке этого примера создается итератор вывода типа `ostreambuf_iterator` для объекта `cout`. Вместо передачи потока вывода можно также непосредственно передать указатель на потоковый буфер. Остальные команды создают объект класса `string` и копируют символы из этого объекта с помощью итератора вывода.

В табл. 15.45 перечислены все операции над итераторами потоковых буферов вывода. Этот интерфейс похож на потоковые итераторы вывода (см. раздел 9.4.3). Кроме того, итератор можно инициализировать буфером, а также проверить возможность записи с помощью функции `failed()`. Если какая-нибудь из предыдущих операций вывода символов завершилась неудачей, то функция `failed()` возвращает значение `true`. В этом случае любые попытки записи с помощью операции = безрезультатны.

Таблица 15.45. Операции над итераторами потоковых буферов вывода

Выражение	Описание
`ostreambuf_iterator<`*char*`>(`*ostream*`)`	Создает итератор потокового буфера вывода для потока *ostream*
`ostreambuf_iterator<`*char*`>(`*buffer_ptr*`)`	Создает итератор потокового буфера вывода для буфера, на который ссылается указатель *buffer_ptr*
**iter*	Фиктивная операция (возвращает *iter*)
iter = *c*	Записывает символ *c* в буфер, вызывая для него `sputc(`*c*`)`
++iter	Фиктивная операция (возвращает *iter*)
iter++	Фиктивная операция (возвращает *iter*)
`failed()`	Проверяет, возможна ли запись с помощью итератора потока вывода

Итераторы потоковых буферов ввода

В табл. 15.46 перечислены все операции над итераторами потоковых буферов ввода. Этот интерфейс похож на потоковые итераторы ввода (см. раздел 9.4.3). Кроме того, итератор можно инициализировать буфером, а также проверить равенство двух итераторов потоковых буферов ввода с помощью функции-члена `equal()`. Два итератора потоковых буферов ввода равны, если они оба установлены в конец потока или ни один из них не установлен в конец потока.

Таблица 15.46. Операции над итераторами потоковых буферов ввода

Выражение	Описание
`istreambuf_iterator<`*char*`>()`	Создает итератор конца потока
`istreambuf_iterator<`*char*`>(`*istream*`)`	Создает итератор потокового буфера ввода для потока *istream*, и может считать первый символ с помощью функции `sgetc()`
`istreambuf_iterator<`*char*`>(`*buffer_ptr*`)`	Создает итератор потокового буфера ввода для буфера, на который ссылается указатель *buffer_ptr* и может считать первый символ с помощью функции `sgetc()`

Окончание табл. 15.46

Выражение	Описание
**iter*	Возвращает текущий символ, считанный ранее функцией `sgetc()` (считывает первый символ, если он не был прочитан конструктором)
++iter	Считывает следующий символ с помощью функции `sbumpc()` и возвращает его позицию
iter++	Считывает следующий символ с помощью функции `sbumpc()`, но возвращает итератор (прокси), для которого оператор `*` возвращает предыдущий символ
iter1`.equal(`*iter2*`)`	Проверяет, равны ли итераторы
iter1`==` *iter2*	Проверяет равенство итераторов *iter1* и *iter2*
iter1`!=` *iter2*	Проверяет неравенство итераторов *iter1* и *iter2*

Немного не ясно, какие объекты класса `istreambuf_iterator` считаются эквивалентными: то ли два объекта типа `istreambuf_iterator`, если оба итератора установлены в конец потока, то ли если ни один из них не установлен в конец потока (совпадают ли при этом буфера не важно). Итератор, установленный в конец потока, может быть получен при создании итератора конструктором по умолчанию. Кроме того, итератор типа `istreambuf_iterator` устанавливается в конец потока данных при попытке вывести итератор за конец потока (т.е. когда функция `sbumpc()` возвращает значение `traits_type::eof`). У такого поведения есть два основных последствия.

1. Диапазон между текущей позицией и концом потока определяется двумя итераторами: `istreambuf_iterator<charT,traits>(`*stream*`)` для текущей позиции и `istreambuf_iterator<charT,traits>()` для конца потока (аргумент *stream* имеет тип `basic_istream<charT,traits>` или `basic_streambuf<charT,traits>`).
2. С помощью итераторов типа `istreambuf_iterator` невозможно создавать подынтервалы.

Пример использования итераторов потоковых буферов

Следующий пример является классическим фильтром, который просто выводит все считанные символы с помощью итераторов потоковых буферов. Он представляет собой измененную версию примера, приведенного в разделе 15.5.3.

```
// io/charcat2.cpp

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

int main()
{
   // итератор потокового буфера для потока cin
   istreambuf_iterator<char> inpos(cin);
```

```
    // итератор конца потока
    istreambuf_iterator<char> endpos;

    // итератор потокового буфера вывода для потока cout
    ostreambuf_iterator<char> outpos(cout);

    // пока итератор ввода является корректным
    while (inpos != endpos) {
        *outpos = *inpos;  // присваиваем его значение итератору вывода
        ++inpos;
        ++outpos;
    }
}
```

Кроме того, можно передать итераторы потоковых буферов алгоритмам, чтобы обрабатывать все символы из потока ввода (полный пример приведен в файле `io/countlines1.cpp`).

```
int countLines (std::istream& in)
{
    return std::count(std::istreambuf_iterator<char>(in),
                      std::istreambuf_iterator<char>(),
                      '\n');
}
```

Пример, в котором все символы из стандартного потока ввода используются для инициализации строки, см. в разделе 14.6.

15.13.3. Пользовательские потоковые буфера

Потоковые буфера предназначены для ввода-вывода. Их интерфейс определяется классом `basic_streambuf<>`. Для символьных типов `char` и `wchar_t` определены специализации `streambuf` и `wstreambuf` соответственно. Эти классы используются как базовые при реализации связи через специальные каналы ввода-вывода. Однако для этого необходимо понимать принципы работы потоковых буферов.

Центральный интерфейс буферов состоит из трех указателей для каждого из двух буферов. Указатели, возвращаемые функциями `eback()`, `gptr()` и `egptr()`, образуют интерфейс буфера чтения. Указатели, возвращаемые функциями `pbase()`, `pptr()` и `epptr()`, образуют интерфейс буфера записи. С этими указателями работают операции чтения и записи, что приводит к соответствующей реакции в канале ввода или вывода. Операция проверяется отдельно для чтения и записи.

Пользовательские потоковые буфера вывода

Буфер, используемый для записи символов, работает с тремя указателями, которые могут быть получены функциями `pbase()`, `pptr()` и `epptr()` (рис. 15.4).

1. `pbase()` ("база вывода") — начало буфера вывода.
2. `pptr()` ("указатель вывода") — текущая позиция записи.
3. `epptr()` ("указатель на конец вывода") — конец буфера вывода. Это значит, что указатель `epptr()` ссылается на позицию, следующую за последним буферизуемым символом.

Символы в диапазоне от pbase() до pptr(), за исключением символа, на который ссылается указатель pptr()), уже записаны, но еще не выведены в соответствующий канал вывода.

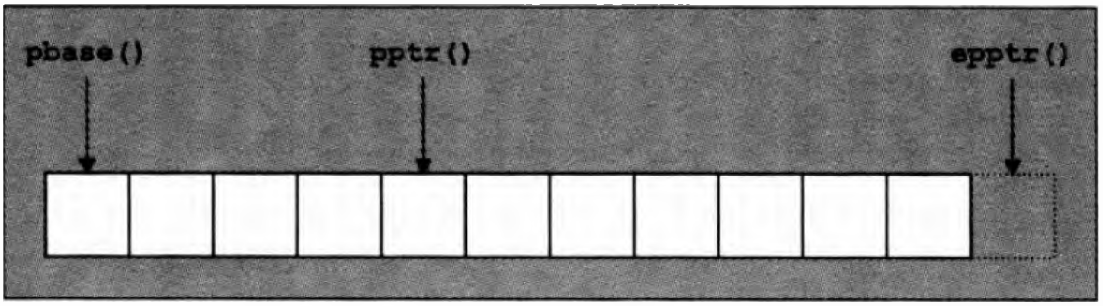


Рис. 15.4. Интерфейс буфера вывода

Символы записываются в буфер функцией-членом sputc(). Символ копируется в текущую позицию записи, если она свободна. Затем указатель на текущую позицию записи инкрементируется. Если буфер полон (т.е. pptr()==epptr()), то содержимое буфера вывода посылается в соответствующий канал вывода с помощью вызова виртуальной функции overflow(). Эта функция предназначена для пересылки символов некоторому внешнему представлению, которое, впрочем, может быть внутренним, как в случае со строковыми потоками. Реализация функции-члена overflow() в базовом классе basic_streambuf возвращает только признак конца файла, означающий, что дальнейшая запись символов невозможна.

Функция-член sputn() позволяет записать сразу несколько символов. Она делегирует это задание виртуальной функции xsputn(), реализацию которой можно оптимизировать для более эффективной записи нескольких символов. Реализация функции-члена xsputn() в классе basic_streambuf вызывает функцию sputc() для каждого символа, поэтому в ее переопределении нет необходимости. Тем не менее во многих случаях запись нескольких символов можно реализовать более эффективно, чем последовательную запись отдельных символов. Таким образом, эту функцию можно использовать для оптимизации обработки последовательностей символов.

Запись в потоковый буфер можно выполнять и без буферизации. Вместо этого символы выводятся сразу же после их получения. В этом случае указателям буфера вывода присваивается значение nullptr (0 или NULL). Конструктор по умолчанию делает это автоматически.

Зная все это, можно реализовать следующий пример, в котором потоковый буфер не использует буферизацию. В нем для каждого символа вызывается функция overflow().

Рассмотрим реализацию этой функции:

```
// io/outbuf1.hpp

#include <streambuf>
#include <locale>
#include <cstdio>

class outbuf : public std::streambuf
{
protected:
   // главная функция вывода
```

```
    // - выводит символы в верхнем регистре
    virtual int_type overflow (int_type c) {
        if (c != EOF) {
            // переводим символы в нижний регистр
            c = std::toupper(c,getloc());

            // записываем символ в стандартный поток вывода
            if (std::putchar(c) == EOF) {
                return EOF;
            }
        }
        return c;
    }
};
```

В данном случае каждый символ, передаваемый в потоковый буфер, записывается с помощью функции `putchar()` из языка C. Однако перед выводом символ преобразуется в верхний регистр функцией `toupper()` (см. раздел 16.4.4). Функция `getloc()` возвращает объект локального контекста, связанный с потоковым буфером (см. раздел 15.8).

В нашем примере буфер вывода реализован специально для типа `char` (тип `streambuf` — это специализация класса `basic_streambuf` для символьного типа `char`). При использовании другого типа символов эту функцию следует реализовать с применением класса свойств символов, рассмотренного в разделе 16.1.4. В этом случае сравнение аргумента `c` и конца файла выполняется иначе. Вместо значения `EOF` должно возвращаться значение `traits::eof()`, а если аргумент `c` равен `EOF`, следует возвращать значение `traits::not_eof(c)` (где `traits` — второй шаблонный аргумент шаблонного класса `basic_streambuf`). Эту функцию можно реализовать следующим образом:

```
// io/outbuf1i18n.hpp

#include <streambuf>
#include <locale>
#include <cstdio>

template <typename charT,
          typename traits = std::char_traits<charT> >
class basic_outbuf : public std::basic_streambuf<charT,traits>
{
  protected:
    // главная функция вывода
    // - выводит символы в верхнем регистре
    virtual typename traits::int_type
            overflow (typename traits::int_type c) {
        if (!traits::eq_int_type(c,traits::eof())) {
            // переводим символы из нижнего в верхний регистр
            c = std::toupper(c,this->getloc());

            // преобразовываем символ в тип char (по умолчанию: '?')
            char cc = std::use_facet<std::ctype<charT>>
                          (this->getloc()).narrow(c,'?');

            // записываем символ в стандартный поток вывода
            if (std::putchar(cc) == EOF) {
```

```
                return traits::eof();
            }
        }
        return traits::not_eof(c);
    }
};

typedef basic_outbuf<char> outbuf;
typedef basic_outbuf<wchar_t> woutbuf;
```

Обратите внимание на то, что в этой программе вызов функции `getloc()` необходимо уточнить выражением `this->`, потому что базовый класс зависит от шаблонного параметра. Кроме того, необходимо сузить символ, прежде чем передавать его функции `putchar()`, потому что функция `putchar()` принимает только аргументы типа `char` (см. раздел 16.4.4).

Применим этот потоковый буфер в следующей программе:

```
// io/outbuf1.cpp

#include <iostream>
#include "outbuf1.hpp"

int main()
{
    outbuf ob;              // создаем специальный потоковый буфер
    std::ostream out(&ob);  // инициализируем потоковый буфер данным буфером

    out << "31 hexadecimal: " << std::hex << 31 << std::endl;
}
```

Результат выглядит следующим образом:

```
31 HEXADECIMAL: 1F
```

Аналогичный подход может использоваться при записи в другие получатели. Например, для инициализации объекта конструктору потокового буфера можно передать дескриптор файла, имя сокетного соединения или два других потоковых буфера, используемых для одновременной записи. Для того чтобы организовать вывод в соответствующий получатель, достаточно реализовать функцию `overflow()`. Кроме того, функцию `xsputn()` следует реализовать для оптимизации вывода в потоковый буфер.

Для того чтобы было проще конструировать потоковые буфера, целесообразно также реализовать специальный класс потока данных, назначение которого сводится к передаче аргументов конструктора соответствующему потоковому буферу. В следующем примере показано, как это делается. В данном случае определяется класс потокового буфера, который инициализируется файловым дескриптором, используемым для записи символов функцией `write()`, низкоуровневой функцией ввода-вывода в системах семейства UNIX). Кроме того, класс, производный от класса `ostream`, определен так, чтобы он поддерживал потоковый буфер, которому передается файловый дескриптор.

```
// io/outbuf2.hpp

#include <iostream>
```

```
#include <streambuf>
#include <cstdio>

// для функции write():
#ifdef _MSC_VER
#include <io.h>
#else
#include <unistd.h>
#endif

class fdoutbuf : public std::streambuf {
   protected:
      int fd; // дескриптор файла
   public:
   // конструктор
   fdoutbuf (int _fd) : fd(_fd) {
   }
protected:
   // записываем один символ
   virtual int_type overflow (int_type c) {
      if (c != EOF) {
          char z = c;
          if (write (fd, &z, 1) != 1) {
              return EOF;
          }
      }
      return c;
   }
   // записываем несколько символов
   virtual std::streamsize xsputn (const char* s,
                                   std::streamsize num) {
      return write(fd,s,num);
   }
};

class fdostream : public std::ostream {
   protected:
      fdoutbuf buf;
   public:
      fdostream (int fd) : std::ostream(0), buf(fd) {
         rdbuf(&buf);
   }
};
```

В данном потоковом буфере также реализована функция `xsputn()`, позволяющая предотвратить вызов функции `overflow()` для каждого символа при отправке в буфер символьной последовательности. Функция `xsputn()` записывает всю последовательность символов в файл, заданный дескриптором `fd`, в рамках одного вызова и возвращает количество успешно выведенных символов.

```
// io/outbuf2.cpp

#include <iostream>
```

```
#include "outbuf2.hpp"
int main()
{
   fdostream out(1); // поток с буфером записи в файловый дескриптор 1

   out << "31 hexadecimal: " << std::hex << 31 << std::endl;
}
```

Эта программа создает поток вывода, инициализируемый файловым дескриптором 1. По действующим правилам этот дескриптор соответствует стандартному каналу вывода. Следовательно, в данном примере символы будут просто направляться в стандартный поток вывода. В аргументе конструктора также могут использоваться другие дескрипторы, например, дескриптор файла или сокета.

При реализации буфер записи должен быть инициализирован с помощью функции `setp()`. Это демонстрируется следующим примером:

```
// io/outbuf3.hpp

#include <cstdio>
#include <streambuf>

// для функции write():
#ifdef _MSC_VER
# include <io.h>
#else
# include <unistd.h>
#endif

class outbuf : public std::streambuf {
   protected:
      static const int bufferSize = 10; // размер буфера данных
      char buffer[bufferSize];          // буфер данных
   public:
      // конструктор
      // - инициализатор буфера данных
      // - на один символ меньше, чтобы при накоплении bufferSize символов
      //   вызывалась функция overflow()
   outbuf() {
      setp (buffer, buffer+(bufferSize-1));
   }

   // деструктор
   // - очистка буфера данных
   virtual ~outbuf() {
      sync();
   }

   protected:
      // выгружаем символы, хранящиеся в буфере
      int flushBuffer () {
      int num = pptr()-pbase();
      if (write (1, buffer, num) != num) {
          return EOF;
      }
      pbump (-num); // соответствующее перемещение указателя вывода
```

```
        return num;
    }

    // буфер полный
    // - записать символ c и все предыдущие символы
    virtual int_type overflow (int_type c) {
        if (c != EOF) {
            // вставляем символ в буфер
            *pptr() = c;
            pbump(1);
        }

        // очищаем буфер
        if (flushBuffer() == EOF) {
            // ОШИБКА
            return EOF;
        }
        return c;
    }

    // синхронизация данных с файлом/ получателем
    // - вывод данных из буфера
    virtual int sync () {
        if (flushBuffer() == EOF) {
            // ОШИБКА
            return -1;
        }
        return 0;
    }
};
```

Этот конструктор инициализирует буфер записи с помощью функции `setp()`.

```
setp (buffer, buffer+(size-1));
```

Буфер записи настроен так, что, когда в нем еще остается место для одного символа, функция `overflow()` уже вызвана. Если функция `overflow()` вызывается с аргументом, не равным `EOF`, то соответствующий символ может быть помещен в текущую позицию записи, поскольку указатель на нее не выходит за пределы указателя на конец буфера. После того как аргумент `overflow()` будет помещен в позицию записи, буфер можно очистить.

Этим занимается функция-член `flushBuffer()`. Она записывает символы в канал стандартного вывода (дескриптор файла 1) с использованием функции `write()`. Для перемещения позиции записи назад, в начало буфера, используется функция-член буфера потока `pbump()`.

Функция `overflow()` вставляет в буфер символ, ставший причиной вызова `overflow()`, если он отличается от значения `EOF`. Затем функция `pbump()` смещает текущую позицию записи, чтобы она отражала новый конец блока буферизованных символов. При этом позиция записи временно смещается за конечную позицию (`epptr()`).

Этот класс также содержит виртуальную функцию `sync()`, предназначенную для синхронизации текущего состояния потокового буфера с соответствующим устройством для хранения данных. Обычно для синхронизации достаточно просто очистить буфер. Для небуферизованных версий переопределять эту функцию не обязательно, поскольку нет буфера, который можно было бы очистить.

Виртуальный деструктор гарантирует вывод данных, остающихся в буфере при его уничтожении.

Эти функции перекрываются для большинства потоковых буферов. Если внешнее представление имеет особую структуру, возможно, придется переопределить дополнительные функции. Например, можно переопределить функции `seekoff()` и `seekpos()` для управления позицией записи.

Пользовательские буфера ввода

В принципе механизм ввода работает так же, как и механизм вывода. Однако для ввода существует дополнительная возможность отмены последнего чтения. Функция `sungetc()`, которая вызывается функцией-членом потока ввода `unget()`, и функция `sputbackc()`, которая вызывается функцией-членом потока ввода `putback()`, используются для восстановления потокового буфера в состоянии перед последним чтением. Кроме того, существует возможность чтения следующего символа без перемещения позиции чтения. Следовательно, при реализации чтения из потокового буфера приходится переопределять больше функций, чем при реализации записи в потоковый буфер.

Буфер, используемый для чтения символов, работает с тремя указателями, которые могут быть получены функциями `eback()`, `gptr()` и `egptr()` (рис. 15.5).

1. `eback()` ("конец области возврата") — это начало буфера ввода, или, как следует из имени, конец области возврата, предназначенной для возврата символов. Если не предпринимать специальных мер, символ можно вернуть только до этой позиции.
2. `gptr()` ("указатель ввода") — это текущая позиция чтения.
3. `egptr()` ("указатель на конец ввода") — конец буфера ввода.

Символы, находящиеся между начальной и конечной позициями, были переданы из внешнего представления в память программы, но еще ожидают обработки.

Одиночные символы можно ввести с помощью функций `sgetc()` и `sbumpc()`, которые отличаются друг от друга тем, что функция `sbumpc()` выполняет инкремент указателя текущей позиции ввода, а функция `sgetc()` — нет. Если буфер будет полностью прочитан (т.е. `gptr()==egptr()`), то доступных символов нет и буфер необходимо заполнять заново, вызывая виртуальную функцию `underflow()`, предназначенную для чтения данных. Если доступных символов нет, функция `sbumpc()` вызывает виртуальную функцию `uflow()`. По умолчанию функция `uflow()` просто вызывает функцию `underflow()`, а затем выполняет инкремент указателя. По умолчанию реализация функции `underflow()` в базовом классе `basic_streambuf` возвращает значение `EOF`. Это значит, что читать символы с помощью стандартной реализации по умолчанию невозможно.

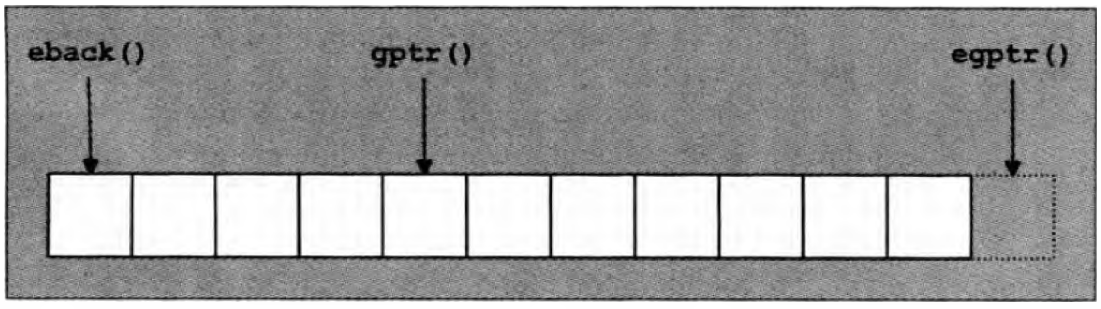


Рис. 15.5. Интерфейс для чтения из потоковых буферов

Функция sgetn() используется для чтения нескольких символов сразу. Она делегирует эту работу виртуальной функции xsgetn(). Реализация функции xsgetn() по умолчанию просто читает символы, вызывая для каждого из них функцию sbumpc(). Аналогично функции xsputn() при записи, функция xsgetn() используется для оптимизации чтения нескольких символов.

В отличие от вывода, для ввода недостаточно переопределить одну функцию. Необходимо либо настроить буфер, либо, по крайней мере, реализовать функции underflow() и uflow(). Это объясняется тем, что функция underflow() не перемещает указатель чтения за текущий символ, однако она может быть вызвана из функции sgetc(). Перемещение к следующему символу приходится выполнять с помощью манипуляций с буфером или вызывая функцию uflow(). В любом случае функция underflow() должна быть реализована для любого потокового буфера, поддерживающего чтение символов. Если реализованы обе функции, underflow() и uflow(), настраивать буфер не обязательно.

Настройку буфера чтения осуществляет функция-член setg(), получающая следующие три аргумента в указанном порядке.

- указатель на начало буфера (eback());
- указатель на текущую позицию чтения (gptr());
- указатель на конец буфера (egptr()).

В отличие от функции setp(), функция setg() вызывается с тремя аргументами, чтобы зарезервировать память для символов, возвращаемых в поток данных. Таким образом, при настройке буфера ввода целесообразно, чтобы хотя бы один символ был уже прочитан, но еще не помещен в буфер.

Как указывалось ранее, символы можно вернуть в буфер чтения с помощью функций sputbackc() и sungetc(). Функция sputbackc() получает в качестве аргумента возвращаемый символ и проверяет, что именно этот символ был прочитан последним. Обе функции выполняют декремент указателя текущей позиции чтения, если это возможно. Очевидно, это возможно только в том случае, если указатель чтения не находится в начале буфера ввода. При попытке вернуть символ, когда достигнуто начало буфера, вызывается виртуальная функция pbackfail(). Переопределяя эту функцию, можно реализовать механизм восстановления прежней позиции чтения даже в этом случае. В базовом классе basic_streambuf соответствующее поведение не определено. Таким образом, на практике возврат на произвольное количество символов невозможен. Для потоков данных, не использующих буферизацию, необходимо реализовать функцию pbackfail(), потому что в общем случае предполагается, что хотя бы один символ может быть возвращен в поток.

После прочтения нового буфера возникает другая проблема: если прежние данные не были сохранены в буфере, возврат в буфер даже одного символа невозможен. Таким образом, реализация функции underflow() часто перемещает несколько последних символов (например, четыре) в начало буфера и присоединяет вновь читаемые символы после них. Это позволяет вернуть хотя бы несколько символов перед тем, как будет вызвана функция pbackfail().

Следующий пример демонстрирует, как может выглядеть подобная реализация. Класс inbuf реализует буфер ввода, рассчитанный на десять символов. Буфер разделяется на две части: область возврата из четырех символов и обычный буфер ввода из шести символов.

```
// io/inbuf1.hpp

#include <cstdio>
```

```
#include <cstring>
#include <streambuf>

// для функции read():
#ifdef _MSC_VER
# include <io.h>
#else
# include <unistd.h>
#endif

class inbuf : public std::streambuf {

   protected:
      // буфер данных:
      // - не более четырех символов в области возврата плюс
      // - до шести символов в обычном буфере ввода
      static const int bufferSize = 10; // размер буфера данных
      char buffer[bufferSize];          // буфер данных

   public:
      // конструктор
      // - инициализирует пустой буфер данных
      // - без области возврата
      // => вызывает функцию underflow()
      inbuf() {
         setg (buffer+4,     // начало буфера возврата
               buffer+4,     // позиция чтения
               buffer+4);    // конечная позиция
   }

   protected:
      // вставляем новые символы в буфер
      virtual int_type underflow () {
         // находится ли позиция чтения перед концом буфера?
         if (gptr() < egptr()) {
             return traits_type::to_int_type(*gptr());
         }

      // обрабатываем размер области возврата
      // - используем количество считанных символов
      // - но не более четырех
      int numPutback;
      numPutback = gptr() - eback();
      if (numPutback > 4) {
          numPutback = 4;
      }

      // копируем четыре ранее считанных символа
      // в буфер возврата (область первых четырех символов)
      std::memmove (buffer+(4-numPutback), gptr()-numPutback,
                    numPutback);

      // читаем новые символы
      int num;
      num = read (0, buffer+4, bufferSize-4);
```

```
        if (num <= 0) {
            // ОШИБКА или EOF
            return EOF;
        }

        // восстанавливаем указатели буфера
        setg (buffer+(4-numPutback), // начало области возврата
              buffer+4,              // позиция чтения
              buffer+4+num);         // конец буфера

        // возвращаем следующий символ
        return traits_type::to_int_type(*gptr());
    }
};
```

Конструктор инициализирует все указатели так, что буфер остается совершенно пустым (рис. 15.6). При попытке прочитать символы из этого буфера вызывается функция `underflow()`, которая всегда используется потоковыми буферами для чтения следующих символов, но сначала она проверяет наличие прочитанных символов в буфере ввода. Если такие символы есть, они перемещаются в область возврата функцией `memcpy()`. В буфере ввода хранятся не более четырех последних символов. Низкоуровневая функция ввода-вывода POSIX `read()` читает следующий символ из стандартного канала ввода. После того как указатели буфера будут настроены в соответствии с новой ситуацией, возвращается первый прочитанный символ.

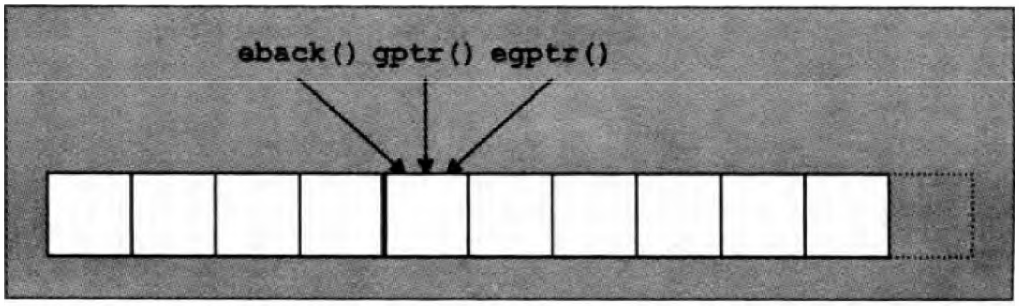


Рис. 15.6. Буфер ввода после инициализации

Например, если при первом вызове функции `read()` были прочитаны символы `'H'`, `'a'`, `'l'`, `'l'`, `'o'` и `'w'`, состояние буфера ввода изменится так, как показано на рис. 15.7. Область возврата остается пустой, потому что буфер заполняется впервые и еще нет символов, подлежащих возврату.

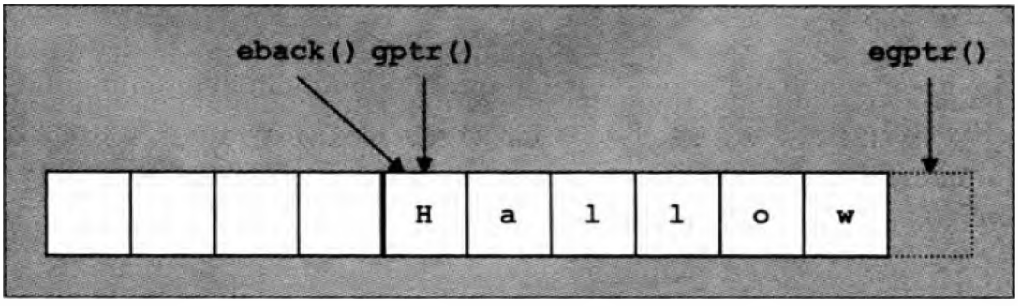


Рис. 15.7. Буфер ввода после чтения символов `H a l l o w`

После извлечения этих символов последние четыре символа перемещаются в область возврата, после чего читаются новые символы. Например, если при следующем вызове `read()` были прочитаны символы `'e'`, `'e'`, `'n'` и `'\n'`, возникнет ситуация, изображенная на рис. 15.8.

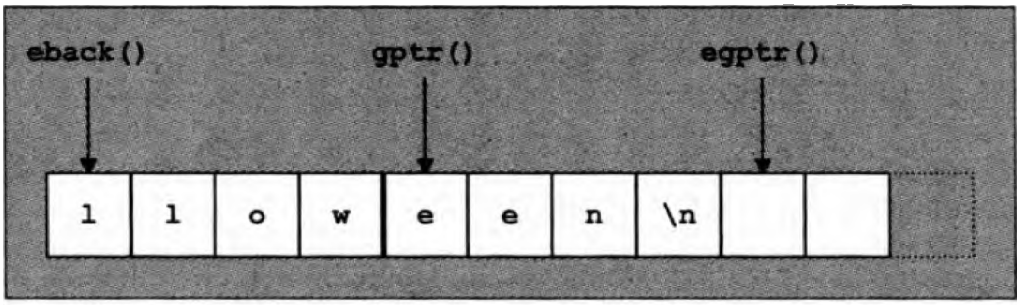


Рис. 15.8. Буфер ввода после чтения следующих четырех символов

Рассмотрим пример использования этого буфера ввода:

```
// io/inbuf1.cpp

#include <iostream>
#include "inbuf1.hpp"

int main()
{
   inbuf ib;               // создаем специальный потоковый буфер
   std::istream in(&ib);   // инициализируем поток ввода с помощью этого буфера

   char c;
   for (int i=1; i<=20; i++) {
        // вводим следующий символ (из буфера)
        in.get(c);

        // выводим этот символ (и очищаем буфер)
        std::cout << c << std::flush;

        // после ввода восьми символов возвращаем в поток два символа
        if (i == 8) {
            in.unget();
            in.unget();
        }
   }
   std::cout << std::endl;
}
```

Эта программа циклически вводит символы и выводит их в поток данных `cout`. После чтения восьмого символа два символа возвращаются в буфер. В результате седьмой и восьмой символы выводятся дважды.

15.14. Проблемы эффективности

В этом разделе рассматриваются вопросы эффективности. Вообще говоря, потоковые классы работают достаточно быстро, но в приложениях, для которых быстродействие ввода-вывода является важным, их можно сделать еще эффективнее.

Одна из проблем быстродействия уже упоминалась в разделе 15.2.3, а именно: в программу должны включаться только заголовочные файлы, абсолютно необходимые для компиляции. В частности, следует избегать включения файла `<iostream>`, если в программе не используются стандартные потоковые объекты.

15.14.1. Синхронизация со стандартными потоками языка C

По умолчанию восемь стандартных потоков данных C++ — четыре символьных потока с однобайтовой кодировкой `cin`, `cout`, `cerr` и `clog`, а также четыре их аналога с расширенной кодировкой — синхронизируются с соответствующими каналами из стандартной библиотеки C: `stdin`, `stdout` и `stderr`. По умолчанию потоки `clog` и `wclog` используют тот же потоковый буфер, что и потоки `cerr` и `wcerr` соответственно. Таким образом, по умолчанию они синхронизируются с потоком `stderr`, хотя в стандартной библиотеке C у этих потоков данных нет прямых аналогов.

В зависимости от реализации синхронизация может приводить к излишним затратам. Например, реализация стандартных потоков данных C++ с использованием стандартных файлов C подавляет буферизацию соответствующих потоковых буферов. Однако буферизация необходима для оптимизации некоторых операций, особенно форматированного чтения (см. раздел 15.14.2). Для того чтобы программист мог переключиться на более эффективную реализацию, в классе `ios_base` определена статическая функция `sync_with_stdio()` (табл. 15.47).

Таблица 15.47. Синхронизация стандартных потоков из языков C++ и C

Статическая функция	Описание
`sync_with_stdio()`	Возвращает информацию о том, синхронизируются ли стандартные объекты потоков данных со стандартными потоками данных C и поддерживается ли параллельная работа
`sync_with_stdio(false)`	Отключает синхронизацию потоков из языков C++ и C (должна вызваться до выполнения любой операции ввода-вывода)

Функция `sync_with_stdio()` получает необязательный булев аргумент, который указывает, нужно ли включать синхронизацию со стандартными потоками данных C. Для того чтобы отключить синхронизацию, функция вызывается с аргументом `false`:

```
std::ios::sync_with_stdio(false); // отключаем синхронизацию
```

Напомним, что синхронизация отключается только до выполнения любой другой операции ввода-вывода. Если это условие не выполнено, последствия от вызова функции зависят от реализации.

Функция `sync_with_stdio()` возвращает значение, использованное при предыдущем вызове. Если ранее функция не вызывалась, она всегда возвращает значение `true`, отражающее состояние по умолчанию для стандартных потоков данных.

После принятия стандарта C++11 отключение синхронизации со стандартными потоками языка C одновременно отключает поддержку параллельной работы. Это позволяет нескольким потокам выполнения использовать стандартный потоковый объект, несмотря на то, что возможно появление промежуточных символов (см. раздел 4.5).

15.14.2. Буферизация в потоковых буферах

Буферизация ввода-вывода оказывает большое влияние на эффективность. Отчасти это объясняется тем, что системные вызовы обычно обходятся относительно дорого, поэтому их по возможности следует избегать. Тем не менее существует и другая, более тонкая причина для буферизации в потоковых буферах C++, по крайней мере, при вводе: функции форматного ввода-вывода обращаются к потокам данных с помощью итераторов потоковых буферов, а операции над итераторами выполняются медленнее операций над указателями. Разница в быстродействии не очень большая, но вполне достаточная для того, чтобы оправдать применение оптимизированных реализаций для часто выполняемых операций, например, форматированного ввода числовых значений. Однако для этого необходима буферизация в потоковых буферах.

Таким образом, весь ввод-вывод осуществляется через потоковые буфера, обеспечивающие механизм буферизации. Однако полагаться только на эту буферизацию недостаточно по трем причинам.

1. Потоки данных без буферизации часто реализуются проще. Если соответствующие потоки данных используются редко или только для вывода, вероятно, буферизация не играет особой роли. При выводе данных разница между итераторами и указателями не столь важна, как при вводе; основная проблема состоит в сравнении итераторов потоковых буферов. Однако, если потоковый буфер используется интенсивно, для него определенно следует реализовать буферизацию.
2. Если установлен флаг `unitbuf`, то поток вывода очищает буфер после выполнения каждой операции вывода. Кроме того, очистка производится манипуляторами `flush` и `endl`. Для оптимального быстродействия желательно избегать всех трех способов. Но при выводе на консоль, например, было бы логично очищать буфер после вывода полных строк. Если вы пишете программу, интенсивно использующую манипуляторы `unitbuf`, `flush` и `endl`, подумайте о реализации специального потокового буфера, который в соответствующий момент вызывает не функцию `sync()`, а другую функцию.
3. Связывание потоков с помощью функции `tie()` (см. раздел 15.12.1) также требует дополнительной очистки потоков. Следовательно, связывание должно применяться только в крайнем случае.

При разработке новых потоковых буферов целесообразно сначала реализовать их без буферизации. Если потоковый буфер окажется узким местом, вы сможете организовать буферизацию, не затрагивая остальные части приложения.

15.14.3. Непосредственное использование потоковых буферов

Все функции-члены классов `basic_istream` и `basic_ostream`, выполняющие чтение или запись символов, работают по одной и той же схеме: сначала конструируется соответствующий объект `sentry`, а затем выполняется операция. Создание объекта `sentry` приводит к очистке буферов потенциально связанных объектов, игнорированию пропусков при вводе и выполнению операций, специфических для конкретных реализаций, например, операций блокировки файлов в средах с параллельным функционированием в многопоточных средах (см. раздел 15.5.4).

При неформатированном вводе-выводе большинство этих операций обычно бесполезно. Исключение составляет операция блокировки, которая может оказаться полезной при работе с потоками в многопоточных средах. Следовательно, при неформатированном вводе-выводе непосредственное использование потоковых буферов обычно более эффективно.

Для поддержки этой возможности для потоковых буферов можно использовать операторы << и >>.

- Передавая оператору << указатель на потоковый буфер, можно вывести все входные данные его устройства. Вероятно, это самый быстрый способ копирования файлов с использованием потоков данных C++. Вот пример такого копирования:

```
// io/copy1.cpp

#include <iostream>
int main ()
{
   // копируем все данные из стандартного потока ввода
   // в стандартный поток вывода
   std::cout << std::cin.rdbuf();
}
```

  Здесь функция `rdbuf()` возвращает буфер потока `cin` (см. раздел 15.12.2). Следовательно, программа копирует все данные из стандартного потока ввода в стандартный поток вывода.

- Передавая оператору >> указатель на потоковый буфер, можно выполнить чтение данных непосредственно в потоковый буфер. Например, копирование всех данных из стандартного потока ввода в стандартный поток вывода можно было бы выполнить следующим образом:

```
// io/copy2.cpp

#include <iostream>
int main ()
{
   // копируем все данные из стандартного потока ввода
   // в стандартный поток вывода
   std::cin >> std::noskipws >> std::cout.rdbuf();
}
```

Обратите внимание на то, что необходимо сбросить флаг `skipws`. В противном случае ведущие пробельные символы будут игнорироваться (см. раздел 15.7.7).

Непосредственная работа с потоковым буфером может быть целесообразной даже при форматированном вводе-выводе. Например, если программа в цикле вводит много числовых значений, может оказаться достаточным сконструировать всего один объект `sentry`, существующий на протяжении всего цикла. В цикле пробельные символы игнорируются вручную, так как использование манипулятора `ws` также привело бы к конструированию объекта `sentry`, а затем фацет `num_get` (см. раздел 16.4.1) выполняет непосредственное чтение числовых значений.

Отметим, что потоковый буфер не обладает собственным состоянием ошибки. Кроме того, он ничего не знает о потоках ввода и вывода, которые могут к нему подключиться. Следовательно, вызов функции `rdbuf()` в коде

```
// копируем содержимое потока in в поток out
out << in.rdbuf();
```

не может изменить код ошибки потока `in` из-за сбоя или обнаружения конца файла.

Глава 16

Интернационализация

По мере развития глобального рынка *интернационализация*, или сокращенно проблема *i18n*,[1] стала играть более важную роль в разработке программного обеспечения. По этой причине в стандартную библиотеку C++ были включены средства для создания интернациональных программ. В основном эти концепции относятся к вводу-выводу и обработке строк. Именно этим концепциям посвящена данная глава. Выражаю глубокую благодарность Дитмару Кюхлю (Dietmar Kьhl), эксперту по вводу-выводу и интернационализации стандартной библиотеки C++, который написал часть этой главы

Стандартная библиотека C++ предоставляет общие средства поддержки национальных стандартов без привязки к конкретным соглашениям. Например, строки не ограничиваются конкретным типом символов и обеспечивают поддержку 16-разрядных азиатских кодировок. При интернационализации программ должны учитываться два связанных друг с другом аспекта.

1. Разные кодировки символов имеют разные свойства, поэтому для работы с ними необходимы гибкие решения таких, например, проблем: что считать буквой или, еще сложнее, какой тип должен использоваться для представления символов. Тип `char` не подходит для представления кодировок, содержащих более 256 символов.
2. Пользователь рассчитывает, что применяемая им программа соответствует национальным и культурным стандартам (например, при форматировании дат, денежных величин, чисел и булевых значений).

Для решения обеих проблем стандартная библиотека C++ предоставляет соответствующие решения.

Механизм интернационализации основан на использовании *объекта локального контекста* (locale objects), представляющего собой расширяемую коллекцию правил, адаптируемых для конкретных национальных стандартов. Локальные контексты уже применялись в языке C для этой цели. В стандарте C++ этот механизм был обобщен и сделан более гибким. Механизм локальных контекстов в языке C++ может использоваться для выполнения любых настроек в зависимости от рабочей среды или предпочтений пользователя. Например, его можно расширить так, чтобы он учитывал единицы измерения, часовые пояса или стандартный размер бумаги.

Многие механизмы интернационализации почти не требуют дополнительного программирования. Например, в потоковом механизме ввода-вывода языка C++ числовые данные форматируются по правилам локального контекста. Программисту остается лишь сообщить классам потоков ввода-вывода, чтобы они учитывали предпочтения пользователя. Кроме автоматического использования объектов локального контекста, программист может напрямую обращаться к ним для форматирования, сравнения, классификации символов и т. д.

---

[1] Термин *i18n* представляет собой сокращение слова *internationalization*, состоящего из буквы *i*, за которым следует 18 символов и буква *n*.

Строки и потоки используют другую концепцию интернационализации — *свойства символов* (character traits). Свойства определяют основные особенности и операции, зависящие от кодировки (такие, как признак "конца файла" или функции сравнения, присваивания и копирования строк).

Последние изменения в стандарте C++11

Стандарт C++98 регламентировал большинство возможностей для локализации библиотеки. Перечислим наиболее важные возможности, добавленные в стандарт C++11.

- Объектам локального контекста и фацетам можно передавать объекты класса `std::string`, а не только `const char*` (см. раздел 16.2.1).
- Добавлено несколько новых манипуляторов: `get_money()`, `put_money()`, `get_time()` и `put_time()` (см. разделы 16.4.3 и 16.4.2).
- Фацет `time_get<>` содержит функцию-член `get()` для полноценного форматирования строк (см. раздел 16.4.3).
- Фацеты для ввода-вывода чисел поддерживают типы `long long` и `unsigned long long`.
- Добавлено новое значение для символьных масок `blank` и соответствующая вспомогательная функция `isblank()` (см. раздел 16.4.4).
- Свойства символов предусмотрены для типов `char16_t` и `char32_t` (см. раздел 16.1.4).
- Новые классы `wstring_convert` и `wbuffer_convert` поддерживают дополнительные преобразования между разными наборами символов (см. раздел 16.4.4).

16.1. Кодирование и наборы символов

На заре компьютерных наук набор символов для представления в компьютере был ограничен буквами английского алфавита. В эру глобализации размер памяти для представления символов достиг 32 бит, что позволяет кодировать более одного миллиона разных символов[2]. Вследствие этого в разных странах и культурных средах появились разные стандарты и подходы для работы с символами.

16.1.1. Многобайтовый текст и текст из широких символов

Существуют два основных подхода к наборам символов, содержащих более 256 символов: многобайтовая и широкая кодировка (так называемые широкие символы — wide characters).

1. В *многобайтовых кодировках* символы представляются переменным количеством байтов. За однобайтовым символом, например, символом из кодировки ISO-Latin-1, может следовать трехбайтовый символ, например, японский иероглиф.

[2] Текущие 32-разрядные наборы символов допускают представление до 0x10FFFF символов, т.е. 1 114 111 значений.

2. В *широких кодировках* символ всегда представляется постоянным количеством байтов независимо от его типа. В обычных кодировках используются от 2 до 4 байтов. Концептуально такие кодировки не отличаются от однобайтовых представлений, для которых достаточно кодировки ISO Latin-1 или даже ASCII.

Многобайтовая кодировка является более компактной, чем широкая. По этой причине именно многобайтовая кодировка обычно применяется для хранения данных вне программ. И наоборот, символы фиксированного размера намного легче обрабатывать, поэтому в программах обычно используется широкая кодировка.

В многобайтовой строке один и тот же байт может представлять как целый символ, так и его часть. В процессе перебора многобайтовой строки каждый байт интерпретируется в соответствии с текущим "состоянием сдвига". В зависимости от значения байта и текущего состояния сдвига байт может представлять определенный символ или изменение текущего состояния сдвига. Многобайтовая строка всегда начинается с определенного исходного состояния сдвига. Например, в исходном состоянии сдвига байты могут представлять символы в кодировке ISO Latin-1 до тех пор, пока не будет обнаружен специальный управляющий перехода. Символ, следующий за этим управляющим символом, идентифицирует новое состояние сдвига. Например, в новом состоянии сдвига байты могут интерпретироваться как арабские символы до тех пор, пока не будет обнаружен следующий управляющий символ.

16.1.2. Разные кодировки символов

Перечислим наиболее важные кодировки символов.

- **US-ASCII** — семибитовая кодировка, стандартизированная в 1963 году для телетайпов и других устройств. Первые 16 значений считаются непечатаемыми символами. К ним относятся такие символы, как перевод каретки, горизонтальная табуляция, возврат на одну позицию со стиранием или сигнал. Этот набор символов служит основой для остальных кодировок, и обычно значения между `0x20` и `0x7F` обозначают одни и те же символы во всех остальных кодировках.
- **ISO-Latin-1**, или **ISO-8859-1** (см. [ISOLatin1]), — восьмибитовая кодировка, стандартизированная в 1987 году, для представления всех символов из языков Западной Европы. Кроме того, этот набор символов служит основой для всех остальных кодировок, и обычно значения между `0x20` и `0x7F`, а также между `0xA0` и `0xFF` обозначают одни и те же символы во всех остальных кодировках.
- **ISO-Latin-9**, или **ISO-8859-15** (см. [ISOLatin9]), — восьмибитовая кодировка, стандартизированная в 1999 году, для улучшенного представления всех символов из языков Западной Европы путем замены редко встречающихся символов знаком евро и другими специальными символами.
- **UCS-2** — шестнадцатибитовая кодировка фиксированного размера, позволяющая представлять 65536 наиболее важных символов из стандартов *Universal Character Set* и *Unicode*.
- **UTF-8** (см. [UTF8]) — многобайтовая кодировка, использующая от одного до восьми *октетов*, состоящих из восьми бит и использующихся для представления всех символов из стандартов *Universal Character Set* и *Unicode*. Широко используется в сети World Wide Web.

- **UTF-16** — многобайтовая кодировка, использующая от одного до двух *кодовых модулей*, состоящих из 16 бит и использующихся для представления всех символов из стандартов *Universal Character Set* и *Unicode*.
- **UCS-4**, или **UTF-32**, — 32-разрядная кодировка фиксированного размера, предназначенная для представления стандартизированных символов из стандартов *Universal Character Set* и *Unicode*.

Отметим, что кодировки UTF-16 и UTF-32 могут иметь *маркер порядка байтов* (*byte order mark — BOM*) в начале каждой символьной последовательности, чтобы отметить используемый порядок следования байтов — *прямой* (по умолчанию) или *обратный*. В качестве альтернативы можно явным образом указывать кодировки UTF-16BE, UTF-16LE, UTF-32BE и UTF-32LE.

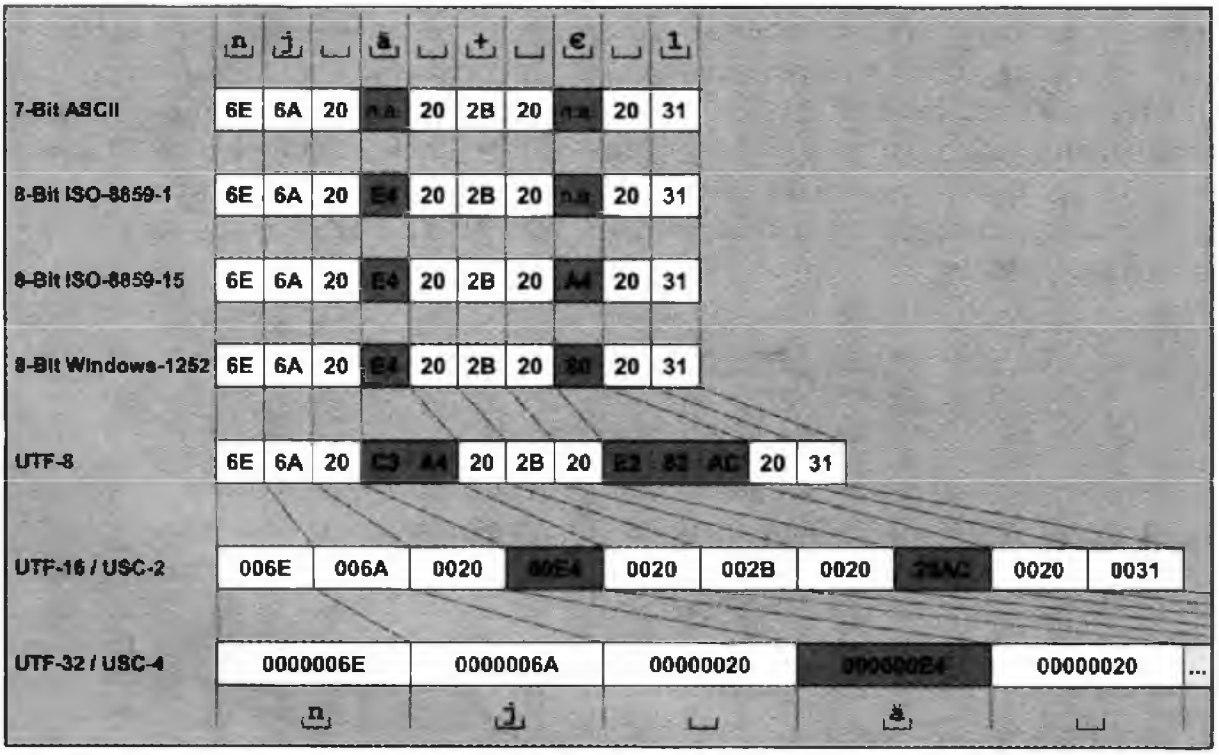


Рис. 16.1. Шестнадцатеричные представления для разных наборов символов

На рис. 16.1 показаны разные шестнадцатеричные представления одной последовательности символов, состоящей из обычных символов ASCII, немецкого умлаута (ä), и символа евро (€). Здесь кодировки UTF-16 и UTF-32 не используют маркеры порядка байтов. Если бы он использовался, то был бы равен `0xFEFF`.

Отметим, что кодировки UTF-16 и UCS-2 почти совпадают вплоть до значения `0xFFFF`. В кодировке UCS-2 нет только очень редких символов, а набор символов UTF-16 использует два элемента кода по 16 бит, учитывая, что кодировка UCS-2 является многобайтовой.

16.1.3. Работа с кодировками в языке C++

В языке C++ существует несколько разных типов для работы с кодировками.

- Тип `char` может использоваться для всех символов, которые можно закодировать 8 битами, например US-ASCII, ISO-Latin-1 и ISOLatin-9. Кроме того, его можно использовать для октета в кодировке UTF-8.

- Тип **char16_t** (предусмотренный стандартом C++11) можно использовать для кодировок UCS-2 и кодирования элемента кода UTF-16.
- Тип **char32_t** (предусмотренный стандартом C++11) можно использовать для кодировок UCS-4/UTF-32.
- Тип **wchar_t** предназначен для значений наибольшего широкого набора символов среди всех поддерживаемых локальных контекстов. Таким образом, он обычно эквивалентен типам `char16_t` или `char32_t`.

Названия всех этих типов представляют собой ключевые слова, поэтому существует возможность перегружать функции для всех этих типов. Отметим, что поддержка типов `char16_t` и `char32_t` является ограниченной. Несмотря на то что свойства символов позволяет работать со строками в кодировке Unicode, перегрузка функций ввода-вывода для этих типов не предусмотрена.

После принятия стандарта C++11 появилась возможность задавать строковые литералы с помощью разных кодировок (см. раздел 3.1.6).

Для поддержки преобразования символов и кодировок в стандартной библиотеке C++ существуют следующие средства.

- Для преобразования типа `string` в `wstring`, и наоборот, можно использовать функции-члены `widen()` и `narrow()` из фацета `ctype<>` (см. раздел 16.4.4). Их можно также использовать для преобразования символов из исходной кодировки в кодировку локального контекста. Обе эти функции используют символьный тип `char`.
- Для преобразования многобайтовых последовательностей в тип `wstring`, и наоборот, можно использовать шаблонный класс `wstring_convert<>` и соответствующие фацеты `codecvt<>` (см. раздел 16.4.4).
- Класс `codecvt<>` (см. раздел 16.4.4) также используется классом `basic_filebuf<>` (см. раздел 15.9.1) для преобразования внутренних и внешних представлений при чтении и записи файлов.
- Для чтения и записи многобайтовых символьных последовательностей можно использовать класс `wbuffer_convert<>` и соответствующие фацеты `codecvt<>` (см. раздел 16.4.4).

16.1.4. Свойства символов

Различия между кодировками оказывают влияние на обработку строк и ввод-вывод. Например, признак конца файла и конкретные особенности сравнения символов могут различаться в зависимости от представления.

Предполагается, что строковые и потоковые классы специализируются встроенными типами, в основном `char` и `wchar_t`, а после принятия стандарта C++11, возможно, `char16_t` и `char32_t`. Интерфейс встроенных типов изменить невозможно, поэтому информация о разных аспектах представления символов выделена в отдельный класс *свойств символов*. Строковые и потоковые классы получают класс свойств в виде шаблонного параметра. По умолчанию в качестве шаблонного параметра задается класс `char_traits`, параметризованный шаблонным параметром, определяющим тип символов строки или потока.

```
namespace std {
   template <typename charT,
             typename traits = char_traits<charT>,
             typename Allocator = allocator<charT>>
             class basic_string;
}

namespace std {
   template <typename charT,
             typename traits = char_traits<charT>>
             class basic_istream;

   template <typename charT,
             typename traits = char_traits<charT>>
             class basic_ostream;
   ...
}
```

Свойства символов имеют тип `char_traits<>`. Этот тип определен в заголовочном файле `<string>` и параметризуется специальным символьным типом.

```
namespace std {
   template <typename charT>
   struct char_traits {
      ...
   };
}
```

Классы свойств определяют все фундаментальные характеристики символьного типа и соответствующие операции, необходимые для реализации строк и потоков как статических компонентов. В табл. 16.1 приведены члены класса `char_traits`.

Таблица 16.1. Члены свойств символов

Выражение	Описание
`char_type`	Символьный тип (шаблонный аргумент класса `char_traits`)
`int_type`	Тип, размеры которого достаточны для представления дополнительного признака конца файла, не используемого для других целей
`pos_type`	Тип, используемый для представления позиций в потоке
`off_type`	Тип, используемый для представления смещений между позициями в потоке
`state_type`	Тип, используемый для представления текущего состояния в многобайтовых потоках
`assign(`*c1*`,`*c2*`)`	Присваивает символ *c2* символу *c1*
`eq(`*c1*`,`*c2*`)`	Проверяет равенство символов *c1* и *c2*
`lt(`*c1*`,`*c2*`)`	Проверяет условие "символ *c1* меньше символа *c2*"
`length(`*s*`)`	Возвращает длину строки *s*

Окончание табл. 16.1

Выражение	Описание
`compare(s1,s2,n)`	Сравнивает до *n* символов строки *s1* и *s2*
`copy(s1,s2,n)`	Копирует *n* символов строки *s2* в строку *s1*
`move(s1,s2,n)`	Копирует *n* символов строки *s2* в строку *s1*, где строки *s1* и *s2* могут перекрываться
`assign(s,n,c)`	Присваивает символ *c* *n* символам строки *s*
`find(s,n,c)`	Возвращает указатель на первый символ в строке *s*, который равен символу *c* или `nullptr`, если среди первых *n* символов такого символа нет
`eof()`	Возвращает признак конца файла
`to_int_type(c)`	Преобразует символ *c* в соответствующее представление типа `int_type`
`to_char_type(i)`	Преобразует представление *i* типа `int_type` в символ (результат преобразования признака конца файла не определен)
`not_eof(i)`	Возвращает значение *i*, если *i* не равно `EOF`; в этом случае возвращается значение, которое зависит от реализации и не совпадает с `EOF`
`eq_int_type(i1,i2)`	Проверяет равенство двух символов *i1* и *i2*, представленных в виде типа `int_type` (символы могут совпадать с `EOF`)

Функции, предназначенные для обработки строк или последовательностей символов, включены только для оптимизации. Их можно реализовать с помощью функций, обрабатывающих отдельные символы. Например, функция `copy()` реализуется с помощью функции `assign()`. Однако для работы со строками можно создать более эффективные реализации.

Количество символов, используемых этими функциями, задаются точно, а не как максимально возможные значения. Иначе говоря, символы завершения строк в этих последовательностях игнорируются.

Последняя группа функций предназначена для работы с символом, служащим признаком конца файла (`EOF`). Этот искусственный символ дополняет кодировку и означает специальную обработку. Для некоторых представлений тип символов может оказаться недостаточно широким для специального символа, поскольку его значение должно отличаться от значений всех "обычных" символов кодировки. В языке C функции для чтения символов возвращали значение типа `int`, а не `char`. В языке C++ этот способ был усовершенствован. В свойствах символов тип `char_type` определяется как тип для представления всех символов, а `int_type` — как тип для представления всех символов и `EOF`. Функции `to_char_type()`, `to_int_type()`, `not_eof()` и `eq_int_type()` определяют соответствующие преобразования и сравнения. Например, возможна ситуация, в которой типы `char_type` и `int_type` являются идентичными по отношению к определенным свойствам символов. Это возможно тогда, когда не все значения типа `char_type` необходимы для представления символов и одно из свободных значений может использоваться для представления признака конца файла.

Для определения позиций и смещения в файлах используются типы `pos_type` и `off_type` соответственно (см. раздел 15.9.4).

В стандартную библиотеку C++ включены специализации char_traits<> для типов char и wchar_t, а после принятия стандарта C++11 и для типов char16_t и char32_t.

```
namespace std {
   template<> struct char_traits<char>;
   template<> struct char_traits<wchar_t>;
   template<> struct char_traits<char16_t>;
   template<> struct char_traits<char32>;
}
```

Специализация для типа char обычно реализуется с помощью глобальных строковых функций языка C, определяемых в заголовочных файлах <cstring> и <string.h>. Реализация может выглядеть следующим образом:

```
namespace std {
   template<> struct char_traits<char> {
      // определения типов:
      typedef char char_type;
      typedef int int_type;
      typedef streampos pos_type;
      typedef streamoff off_type;
      typedef mbstate_t state_type;

      // функции:
      static void assign(char& c1, const char& c2) {
         c1 = c2;
      }
      static bool eq(const char& c1, const char& c2) {
         return c1 == c2;
      }

      static bool lt(const char& c1, const char& c2) {
         return c1 < c2;
      }

      static size_t length(const char* s) {
         return strlen(s);
      }

      static int compare(const char* s1, const char* s2, size_t n) {
         return memcmp(s1,s2,n);
      }

      static char* copy(char* s1, const char* s2, size_t n) {
         return (char*)memcpy(s1,s2,n);
      }

      static char* move(char* s1, const char* s2, size_t n) {
         return (char*)memmove(s1,s2,n);
      }

      static char* assign(char* s, size_t n, char c) {
         return (char*)memset(s,c,n);
      }
```

```
    static const char* find(const char* s, size_t n,
                            const char& c) {
       return (const char*)memchr(s,c,n);
    }

    static int eof() {
       return EOF;
    }

    static int to_int_type(const char& c) {
       return (int)(unsigned char)c;
    }

    static char to_char_type(const int& i) {
       return (char)i;
    }

    static int not_eof(const int& i) {
       return i!=EOF ? i : !EOF;
    }

    static bool eq_int_type(const int& i1, const int& i2) {
       return i1 == i2;
    }
  };
}
```

Реализация пользовательского класса свойств, позволяющего обрабатывать строки без учета их регистра, приведена в разделе 13.2.15.

16.1.5. Интернационализация специальных символов

Осталось рассмотреть один вопрос: как интернационализируются специальные символы, например, символ перехода на новую строку? Для этой цели класс `basic_ios` содержит функции `widen()` и `narrow()`. Таким образом, символ перехода на новую строку в кодировке, соответствующей потоку `strm`, может быть записан следующим образом:

```
strm.widen('\n') // интернационализированный символ перехода на новую строку
```

Символ завершения строки в этой же кодировке можно создать так:

```
strm.widen('\0') // интернационализированный символ завершения строки
```

Использование этих символов показано в примере реализации манипулятора в разделе 15.6.2.

Функции `widen()` и `narrow()` используют объект локального контекста, точнее — фацет `ctype` этого объекта. Этот фацет предназначен для преобразования всех символов типа `char` в другое представление. Он описан в разделе 16.4.4. Например, следующее выражение преобразует символ `c` типа `char` в объект типа `char_type` с использованием объекта локального контекста `loc`.

```
std::use_facet<std::ctype<charType>>(loc).widen(c)
```

Подробности использования объектов локального контекста и их фацетов описаны в следующих разделах.

16.2. Концепция локального контекста

Общепринятый подход к интернационализации основан на использовании специальных сред, называемых *локальными контекстами* и инкапсулирующих национальные или культурные стандарты. Именно этот подход используется в языке C. Таким образом, с точки зрения интернационализации *локальный контекст* представляет собой набор параметров и функций, обеспечивающих поддержку национальных или культурных стандартов. В соответствии с соглашениями X/Open[3] локальный контекст задается переменной окружения `LANG`. В зависимости от локального контекста выбираются разные форматы чисел с плавающей точкой, дат, денежных сумм и т. д.

Обычно локальный контекст определяется строкой в формате

язык[_*зона*[.*код*]][@*модификатор*]

где

- параметр *язык* обозначает естественный язык, например английский или немецкий. Обычно он представляет собой строку из двух букв в нижнем регистре, например `en` или `de`;
- параметр *зона* задает страну, географический регион или культуру, в которой используется этот язык. Обычно он представляет собой строку из двух букв в верхнем регистре, например `US` или `DE`;
- параметр *код* определяет кодировку символов. Например, он важен для азиатских стран, где для одного и того же набора символов могут использоваться разные кодировки. Примеры: `utf8`, `ISO-8859-1`, `eucJP`;
- параметр *модификатор* на некоторых платформах позволяет задавать дополнительные модификации, например, `@euro` для использования символов евро или `@phone` для сортировки телефонной записной книжки.

В табл. 16.2 приведены некоторые типичные определения локальных контекстов, в частности, для систем POSIX. Однако следует помнить, что эти строки *не переносимы*. Фактически в локальных контекстах используется несколько стандартов и специальных значений. Например, для использования немецкого локального контекста с набором символов ISO-Latin1, включая символ евро, можно задать контексты `de_DE.ISO-8859-15` или `de_DE@euro`, или `deu_deu.1252`, или `deu_germany`, или просто `german`. (Отметим: из-за использования разных кодировок символ евро может иметь разные целочисленные значения.) Перечислим некоторые полезные источники информации о локальных контекстах.

- Для параметра *язык* спецификации ISO639 (см. [ISO639:LangCodes]) определяют двухбуквенные акронимы, такие как `en` и `de`, обычно используемые в средах POSIX, и трехбуквенные акронимы, такие как `eng` и `deu`, обычно поддерживаемые платформами Windows.

[3] POSIX и X/Open — стандарты интерфейсов операционных систем.

- Для параметра *зона* спецификации ISO3166 (см. [ISO3166:CodeTab]) определяют двухбуквенные акронимы, такие как `US` или `DE`, обычно используемые средами POSIX. Отметим, что система Windows использует для зон разные коды.
- В спецификациях [VisualC++Locales] описаны параметры *язык*, *зона* и *код* для платформ Windows, которые используют немного разные кодировки.

Таблица 16.2. Выбор названия для локального контекста

Локальный контекст	Описание
`C`	По умолчанию: ANSI-C соглашения (английский язык, 7 бит)
`de_DE`	Немецкий язык в Германии
`de_DE.ISO-8859-1`	Немецкий язык в Германии с кодировкой ISO-Latin-1
`de_DE.utf8`	Немецкий язык в Германии с кодировкой UTF-8
`de_AT`	Немецкий язык в Австрии
`de_CH`	Немецкий язык в Швейцарии
`en_US`	Английский язык в США
`en_GB`	Английский язык в Великобритании
`en_AU`	Английский язык в Австралии
`en_CA`	Английский язык в Канаде
`fr_FR`	Французский язык во Франции
`fr_CH`	Французский язык в Швейцарии
`fr_CA`	Французский язык в Канаде
`ja_JP.jis`	Японский язык в Японии с кодировкой *Japanese Industrial Standard (JIS)*
`ja_JP.sjis`	Японский язык в Японии с кодировкой *Shift JIS*
`ja_JP.ujis`	Японский язык в Японии с кодировкой *UNIXized JIS*
`ja_JP.EUC`	Японский язык в Японии с кодировкой *Extended UNIX Code*
`ko_KR`	Корейский язык в Корее
`zh_CN`	Китайский язык в Китае
`zh_TW`	Китайский язык в Тайване
`lt_LN.bit7`	ISO-Latin, 7 бит
`lt_LN.bit8`	ISO-Latin, 8 бит
`POSIX`	Спецификации POSIX (английский язык, 7 бит)

Несмотря на отсутствие стандартизации названий локальных контекстов, особых проблем, как правило, не возникает, потому что информация локального контекста в той или иной форме предоставляется пользователем. Обычно программа просто извлекает информацию из переменных окружения или базы данных и определяет требуемый локальный контекст. Таким образом, определение правильного названия локального контекста поручается пользователю. Имя локального контекста "зашивается" в программе, только если она всегда использует конкретный локальный контекст. Обычно в таких случаях достаточно локального контекста `C`, который заведомо поддерживается всеми реализациями.

В следующем разделе будет показано, как работать с разными локальными контекстами в программе на языке C++. Кроме того, в нем описаны *фацеты* локальных контекстов, позволяющие использовать специальные варианты форматирования.

Кроме того, в языке C существует механизм для работы с кодировками, содержащими более 256 символов. Этот механизм основан на использовании символьных типов `wchar_t`, представляющих собой один из целочисленных типов с языковой поддержкой широких констант и широких строковых литералов. Помимо этого поддерживаются только функции преобразования между широкой и обычной кодировками. Этот подход также был реализован в языке C++ с помощью символьного типа `wchar_t`, который, в отличие от языка C, является самостоятельным типом. По уровню библиотечной поддержки язык C++ превосходит язык C, потому что практически все возможности типа `char` доступны и для типа `wchar_t`, а также любого другого типа, который может использоваться в качестве символьного типа.

После принятия стандарта C++11 появилась поддержка типов `char16_t` и `char32_t`. Однако она осуществляется не с помощью библиотеки. Например, для этих типов нет стандартных объектов потоков ввода-вывода, в то время как существует поток `wcout`, представляющий собой аналог потока `cout`.

16.2.1. Использование локальных контекстов

Перевода текстовых сообщений обычно недостаточно для полноценной интернационализации. Например, существуют разные стандарты представления чисел, денежных величин или дат, которые необходимо учитывать. Кроме того, функции, работающие с буквами, должны зависеть от кодировок, чтобы гарантировать правильное представление всех символов данного языка.

В соответствии со стандартами POSIX и X/Open в языке C эта возможность уже существовала благодаря функции `setlocale()`. Изменение локального контекста влияло на работу функций, выполняющих распознавание и обработку символов, таких как `isupper()` и `toupper()`, а также функций ввода-вывода, таких как `printf()`.

Однако подход, принятый в языке C, имеет несколько ограничений. Поскольку локальный контекст является глобальным свойством, одновременное использование нескольких локальных контекстов, например, при чтении чисел с плавающей точкой по-английски и вывод их по-немецки, либо невозможно, либо требует относительно больших усилий. Кроме того, эти локальные контексты невозможно расширить. Они обладают только теми возможностями, которые заложены в них при реализации. Если потребуется адаптировать к национальным стандартам какую-нибудь новую возможность, то придется использовать другой механизм. Наконец, в языке C невозможно определить новые локальные контексты для поддержки специальных культурных стандартов.

Стандартная библиотека C++ решает все эти проблемы на основе объектно-ориентированного подхода. Прежде всего, детали локального контекста инкапсулируются в объекте типа `locale`, что позволяет использовать несколько локальных контекстов одновременно. Операции, зависящие от локальных контекстов, настраиваются на применение соответствующего объекта. Например, для каждого потока ввода-вывода можно задать объект локального контекста, который используется функциями-членами потока для адаптации к соответствующим стандартам. Это продемонстрировано в следующем примере:

```
// i18n/locl.cpp

#include <iostream>
```

```
#include <locale>
#include <exception>
#include <cstdlib>
using namespace std;

int main()
{
   try {
      // используем классический локальный контекст языка C
      // для чтения данных и стандартного потока ввода
      cin.imbue(locale::classic());

      // используем немецкий локальный контекст для записи данных
      // в стандартный поток вывода
      // - используем разные имена локальных контекстов
      //   для систем Windows and POSIX
#ifdef _MSC_VER
      cout.imbue(locale("deu_deu.1252"));
#else
      cout.imbue(locale("de_DE"));
#endif

      // вводим и выводим значения с плавающей точкой в цикле
      cout << "input floating-point values (classic notation): " << endl;
      double value;
      while (cin >> value) {
            cout << value << endl;
      }
   }
   catch (const std::exception& e) {
      cerr << "Exception: " << e.what() << endl;
      return EXIT_FAILURE;
   }
}
```

Первое выражение `imbue()` связывает "классический" локальный контекст языка C со стандартным каналом ввода.

```
cin.imbue(locale::classic()); // используем классический локальный
                              // контекст языка C
```

В классическом локальном контексте форматирование чисел и дат, классификация символов и т. д. выполняются так же, как в исходном языке C. Следующее выражение возвращает соответствующий объект класса `locale`:

```
std::locale::classic()
```

Следующее выражение приводит к тому же результату:

```
std::locale("C")
```

Последнее выражение создает объект `locale` с заданным именем. Имя `"C"` — это специальное имя, которое должно поддерживаться реализацией языка C++. Поддержка

других локальных контекстов не гарантирована, хотя предполагается, что язык C++ может поддерживать другие локальные контексты.

В качестве альтернативы можно было бы использовать конструктор по умолчанию класса `locale`, который инициализирует локальный контекст в соответствии с глобальным контекстом программы. По умолчанию им является классический локальный контекст языка C.

```
cin.imbue(locale()); // используем глобальный контекст
                     // (по умолчанию – классический локальный объект языка C)
```

В качестве имени локального контекста может использоваться пустая строка. В этом случае используется "естественный" локальный контекст, соответствующий окружению программы.

```
cin.imbue(locale("")); // использовать локальный контекст,
                       // соответствующий окружению
```

Следующие выражения `imbue()` связывают немецкий локальный контекст со стандартным каналом вывода, используя в системе POSIX акроним `de_DE`, а в среде Windows — акроним `deu_deu.1252`[4]:

```
#ifdef _MSC_VER
   cout.imbue(locale("deu_deu.1252"));
#else
   cout.imbue(locale("de_DE"));
#endif
```

Разумеется, эти директивы выполняются успешно только в том случае, если система поддерживает данный локальный контекст. Если при создании объекта локального контекста будет указано имя, которое реализации неизвестно, генерируется исключение `runtime_error`. По этой причине этот вызов заключен в раздел `try-catch`.

Если все хорошо, то входные данные читаются по классическим правилам языка C, а выходные данные выводятся по немецким стандартам. Тогда цикл читает числа с плавающей точкой, записанные в английском формате:

```
47.11
```

и выводит их в немецком формате:

```
47,11
```

(В немецком языке вместо десятичной точки используется запятая[5].)

Как правило, программы не определяют заранее локальный контекст, кроме случаев, когда чтение и запись данных производятся в фиксированном формате. Вместо этого локальный контекст определяется с помощью переменной окружения `LANG`. Кроме того, имя локального контекста может считываться из других источников. Эта возможность продемонстрирована в следующем примере:

---

[4] Акронимы "de" и "deu" означают "Deutschland" — Германия по-немецки.

[5] Тот же эффект даст применение русского контекста (`rus_rus.1251` в Windows, `ru_RU` в POSIX). — *Примеч. консульт.*

```
// i18n/loc2.cpp

#include <iostream>
#include <locale>
#include <string>
#include <cstdlib>
#include <exception>
using namespace std;

int main()
{
   try {
      // создаем локальный контекст по умолчанию
      // с помощью пользовательского окружения
      locale langLocale("");

      // связываем его со стандартным каналом вывода
      cout.imbue(langLocale);

      // анализируем имя локального контекста, чтобы выяснить,
      // следует ли использовать немецкий локальный контекст
      cout << langLocale.name() << endl;
      bool isGerman = (langLocale.name().substr(0,2) == "de" ||
                       langLocale.name().substr(0,3) == "ger" ||
                       langLocale.name().substr(0,3) == "Ger");

      // считываем локальный контекст для ввода
      cout << (isGerman ? "Sprachumgebung fuer Eingaben: "
               : "Locale for input: ") << endl;
      string s;
      cin >> s;
      if (!cin) {
          if (isGerman) {
              cerr << "FEHLER beim Einlesen der Sprachumgebung"
                   << endl;
          }
          else {
              cerr << "ERROR while reading the locale" << endl;
          }
          return EXIT_FAILURE;
      }
      locale cinLocale(s);  // создаем локальный контекст по строке (C++11)
                            // и связываем его со стандартным каналом ввода
      cin.imbue(cinLocale);

      // считываем и выводим числа с плавающей точкой в цикле
      cout << (isGerman ? "Gleitkommawerte: "
               : "Floating-point values: ") << endl;
      double value;
      while (cin >> value) {
          cout << value << endl;
      }
   }
   catch (const std::exception& e) {
      cerr << "Exception: " << e.what() << endl;
```

```
        return EXIT_FAILURE;
    }
}
```

В этом примере следующий оператор создает объект класса `locale`:

```
locale langLocale("");
```

Передача пустой строки вместо имени локального контекста имеет особый смысл: она обозначает "естественный" локальный контекст, который определяется пользовательским окружением (как правило, переменной окружения `LANG`). Этот локальный контекст связывается со стандартным входным потоком данных с помощью оператора

```
cout.imbue(langLocale);
```

Следующее выражение извлекает имя локального контекста и возвращает объект типа `string` (см. главу 13):

```
langLocale.name()
```

Следующие операторы создают локальный контекст по имени, считанному из стандартного потока ввода:

```
string s;
cin >> s;
...
locale cinLocale(s); // создаем локальный контекст по строке (C++11)
```

Для этого из стандартного потока ввода считывается слово, которое передается конструктору как аргумент. До принятия стандарта C++11 конструктор класса `locale` получал только аргументы типа `const char*`, поэтому приходилось писать

```
locale cinLocale(s.c_str()); // создание локального контекста до принятия
                             // стандарта C++11
```

Если чтение завершилось неудачно, то в потоке ввода устанавливается флаг `ios_base::failbit`, который затем проверяется и обрабатывается программой.

```
if (!cin) {
    if (isGerman) {
        cerr << "FEHLER beim Einlesen der Sprachumgebung"
             << endl;
    }
    else {
        cerr << "ERROR while reading the locale" << endl;
    }return EXIT_FAILURE;
}
```

Если строка имеет некорректное значение для создания локального контекста, то генерируется исключение `runtime_error`.

Если программа должна соблюдать национальные стандарты, необходимо использовать соответствующие объекты локальных контекстов. Статическая функция-член `global()` класса `std::locale` инсталлирует новый глобальный контекст. Этот объект используется по умолчанию функциями, которым при вызове локальный контекст передается как

необязательный аргумент. Если у объекта локального контекста, назначенного функцией-членом `global()`, есть имя, функции языка С, предназначенные для работы с локальным контекстом, будут реагировать соответственно. Если локальный контекст не имеет имени, то результаты вызова функций языка С зависят от реализации.

Рассмотрим пример установки глобального контекста, зависящего от окружения, в котором выполняется программа:

```
// создаем объект локального контекста в зависимости от окружения
// и устанавливаем его как глобальный
std::locale::global(std::locale(""));
```

Помимо прочего, этот оператор регистрирует выполняемые функции языка С. Иначе говоря, функции языка С работают так, будто был выполнен следующий вызов:

```
std::setlocale(LC_ALL,"")
```

Однако настройка глобального контекста не заменяет локальные контексты, уже хранящиеся в объектах. Она лишь модифицирует объект локального контекста, копируемый при создании контекста конструктором по умолчанию. Например, вызов функции-члена `locale::global()` не влияет на объекты локальных контекстов, хранящиеся в объектах потоков. Если необходимо, чтобы существующий поток использовал конкретный локальный контекст, следует связать его с этим контекстом с помощью функции `imbue()`.

Глобальный контекст используется при создании объектов локального контекста конструктором по умолчанию. В этом случае новый локальный контекст представляет собой копию глобального контекста на момент его создания. Следующие три оператора назначают локальный контекст по умолчанию для стандартных потоков.

```
// Глобальный контекст для потоков
std::cin.imbue(std::locale());
std::cout.imbue(std::locale());
std::cerr.imbue(std::locale());
```

При использовании локальных контекстов в языке C++ важно помнить, что они слабо связаны с локальными контекстами в языке С. Между этими механизмами существует только одна связь: единый локальный контекст в языке С изменяется при создании глобального именованного контекста C++. В принципе нельзя предполагать, что функции языков С и C++ работают в одних и тех же локальных контекстах.

16.2.2. Фацеты

Аспекты национальных стандартов распределены на несколько соответствующих объектов. Объект, предназначенный для учета отдельного аспекта интернационализации, называется *фацетом* (facet). Объект локального контекста используется как контейнер для различных фацетов. Для доступа к отдельному аспекту локального контекста тип соответствующего фацета используется как индекс. Он явным образом передается как шаблонный аргумент функции `use_facet()` для обращения к требуемому фацету. Например, следующее выражение обращается к фацету типа `numpunct<>`, специализированному символьным типом `char` объекта локального контекста `loc`:

```
std::use_facet<std::numpunct<char>>(loc)
```

Каждый тип фацета определяется в виде класса, описывающего некоторые сервисы. Например, тип фацета numpunct<> предоставляет сервис форматирования числовых и булевых величин. Так, следующее выражение возвращает строку, используемую для представления значения true в локальном контексте loc:

```
std::use_facet<std::numpunct<char>>(loc).truename()
```

Отметим, что функция use_facet() возвращает ссылку на объект, который остается корректным только на время существования объекта локального контекста. Таким образом, следующий код приводит к неопределенному поведению, потому что ссылка fac после выполнения первого выражения становится недействительной:

```
const numpunct<char>& fac = use_facet<numpunct<char>>(locale("de"));
cout << "true in German: " << fac.truename() << endl; // ОШИБКА
```

В табл. 16.3 перечислены фацеты, поддерживаемые стандартной библиотекой C++. Каждый фацет относится к определенной категории. Эти категории используются некоторыми конструкторами для создания новых объектов локальных контекстов в виде комбинации других объектов.

Таблица 16.3. Типы фацетов, определенные в стандартной библиотеке C++

Категория	Тип фацета	Предназначение
numeric	num_get<>()	Ввод чисел
	num_put<>()	Вывод чисел
	numpunct<>()	Символы, используемые для ввода и вывода чисел
monetary	money_get<>()	Ввод денежных величин
	money_put<>()	Вывод денежных величин
	moneypunct<>()	Символы, используемые для ввода и вывода денежных величин
time	time_get<>()	Ввод времени и даты
	time_put<>()	Вывод времени и даты
ctype	ctype<>()	Информация о символе (toupper(), isupper())
	codecvt<>()	Преобразования между разными кодировками символов
collate	collate<>()	Контекстное сравнение строк
messages	messages<>()	Чтение строковых сообщений

Пользователь может определять собственные версии фацетов для создания специализированных локальных контекстов. В следующем примере показано, как это делается. Сначала в нем определяется фацет для использования немецких строковых представлений булевых величин:

```
class germanBoolNames : public std::numpunct_byname<char> {
  public:
    germanBoolNames (const std::string& name)
        : std::numpunct_byname<char>(name) {
    }
```

```
  protected:
     virtual std::string do_truename () const {
         return "wahr";
     }
     virtual std::string do_falsename () const {
         return "falsch";
     }
};
```

До принятия стандарта C++11 конструктор должен был иметь аргумент `name` с типом `const char*`.

Класс `germanBoolNames` является производным от класса `numpunct_byname<>`, определенного в стандартной библиотеке C++. Этот класс определяет свойства пунктуации локального контекста, применяемые при числовом форматировании. Использование базового класса `numpunct_byname<>` вместо `numpunct<>` позволяет модифицировать функции-члены класса, которые не переопределяются явным образом. Значения, возвращаемые этими функциями-членами, зависят от имени, переданного как аргумент конструктора. Если бы в качестве базового использовался класс `numpunct<>`, то поведение этих функций было бы фиксированным. Однако класс `germanBoolNames` переопределяет две функции, определяющие текстовое представление значений `true` и `false`.

Для того чтобы использовать этот фацет в локальном контексте, необходимо создать новый объект локального контекста при помощи специального конструктора класса `std::locale`. В первом аргументе этого конструктора передается объект локального контекста, а во втором — указатель на фацет. Созданный локальный контекст идентичен первому аргументу во всем, кроме фацета, переданного во втором аргументе. Этот фацет инсталлируется во вновь созданном объекте контекста после копирования первого аргумента.

```
std::locale loc (std::locale(""), new germanBoolNames(""));
```

Выражение `new` создает фацет, инсталлируемый в новом локальном контексте. Таким образом, фацет регистрируется в объекте `loc` для создания специализированной версии функции `locale("")`. Поскольку объекты локального контекста модифицировать невозможно, для инсталляции нового фацета в локальном контексте необходимо создать новый объект локального контекста. Этот новый объект используется так же, как любой другой объект локального контекста. Например, операторы

```
std::cout.imbue(loc);
std::cout << std::boolalpha << true << std::endl;
```

выводят следующий результат:

```
wahr
```

(Полный пример содержится в файле `i18n/germanbool.cpp`.)

Можно также создать абсолютно новый фацет. В этом случае с помощью функции `has_facet()` можно узнать, был ли этот новый фацет зарегистрирован для заданного объекта локального контекста.

16.3. Подробное описание объекта локального контекста

Локальный контекст в языке C++ представляет собой неизменяемый контейнер, содержащий фацеты. Он определен в заголовочном файле <locale>.

Одно из характерных свойств локального контекста заключается в специальном механизме доступа к объектам, хранящимся в контейнере. Доступ к фацету локального контекста обеспечивается с помощью индекса, представляющего собой тип фацета. Поскольку каждый фацет имеет свой собственный интерфейс и свое предназначение, желательно, чтобы функция, осуществляющая доступ к локальному контексту, возвращала значение, имеющее тип, соответствующий индексу. Именно так происходит индексирование по типу фацета. Использование типа фацета как индекса дает дополнительное преимущество для обеспечения интерфейса, безопасного с точки зрения типов.

Локальный контекст является неизмененным. Это значит, что фацеты, хранящиеся в локальном контексте, не могут изменяться, за исключением ситуаций, когда локальные контексты присваиваются друг другу. Комбинируя локальные контексты и фацеты, можно создавать новые локальные контексты. Конструкторы локальных контекстов перечислены в табл. 16.4.

Таблица 16.4. Конструирование и присваивание локальных контекстов

Выражение	Описание
`locale()`	Конструктор по умолчанию; создает копию текущего глобального контекста
`locale("")`	Создает естественный локальный контекст в соответствии с окружением
`locale(`*name*`)`	Создает локальный контекст по строке *name*
`locale(`*loc*`)`	Копирующий конструктор; создает копию локального контекста *loc*
`locale(`*loc1*`,`*loc2*`,`*cat*`)`	Создает копию локального контекста *loc1*, причем все фацеты из категории *cat* заменяются фацетами из локального контекста *loc2*
`locale(`*loc*`,`*name*`,`*cat*`)`	Эквивалент `locale(`*loc*`, locale(`*name*`) ,`*cat*`)`
`locale(`*loc*`,`*fp*`)`	Создает копию локального контекста *loc* и инсталлирует фацет, на который ссылается указатель
loc1`.combine<`*F*`>(`*loc2*`)`	Создает и возвращает копию локального контекста *loc1*, но с фацетом типа *F*, полученным от локального контекста *loc2*
loc1 = *loc2*	Присваивает локальный контекст *loc2* локальному контексту *loc1*

Почти все конструкторы создают копию другого локального контекста. Простое копирование локального контекста считается относительно дешевой операцией, которая сводится к установке указателя и увеличению счетчика ссылок. Создание модифицированного локального контекста — более трудоемкая операция, поскольку необходимо модифицировать счетчики ссылок для всех фацетов, хранящихся в локальном контексте. Хотя стандарт не гарантирует эффективность этой операции, весьма вероятно, что во всех реализациях копирование локальных контекстов будет производиться достаточно эффективно.

Два конструктора из табл. 16.4 получают имена локальных контекстов в виде аргументов. Передаваемые имена не стандартизированы, за исключением имени C (см. раздел 16.2). До принятия стандарта C++11 *имя* должно было задаваться С-строкой.

Функция-член `combine()` требует дополнительных пояснений, потому что она представляет собой шаблонную функцию с явно заданным шаблонным аргументом. Тип ее шаблонного аргумента не выводится неявно, поскольку не существует типа, из которого его можно было бы вывести. Вместо этого шаблонный аргумент (в данном случае тип *F*) задается явно.

Обе функции для обращения к фацетам объекта локального контекста используют одинаковый механизм (табл. 16.5). Главное их отличие заключается в том, что эти функции являются глобальными и шаблонными.

Таблица 16.5. Доступ к фацетам

Выражение	Описание
`has_facet<`*F*`>(`*loc*`)`	Возвращает значение `true`, если фацет типа *F* хранится в локальном контексте *loc*
`use_facet<`*F*`>(`*loc*`)`	Возвращает ссылку на фацет типа *F*, хранящийся в локальном контексте *loc*

Функция `use_facet()` возвращает ссылку на фацет. Тип этой ссылки соответствует типу, переданному явно в шаблонном аргументе. Если в локальном контексте, заданном аргументом, нет соответствующего фацета, то функция генерирует исключение `bad_cast`. Для того чтобы узнать, содержится ли в заданном локальном контексте конкретный фацет, следует использовать функцию `has_facet()`.

Остальные операции над локальными контекстами перечислены в табл. 16.6.

Таблица 16.6. Операции над локальными контекстами

Выражение	Описание
loc`.name()`	Возвращает имя локального контекста *loc* как объект класса `string`
loc1 `==` *loc2*	Возвращает значение `true`, если *loc1* и *loc2* — идентичные локальные контексты
loc1 `!=` *loc2*	Возвращает значение `true`, если *loc1* и *loc2* — разные локальные контексты
loc`(`*str1*`,` *str2*`)`	Возвращает булев результат сравнения строк *str1* и *str2* в лексикографическом порядке (признак того, что *str1* меньше *str2*)
`locale::classic()`	Возвращает `locale("C")`
`locale::global(`*loc*`)`	Создает глобальный контекст *loc* и возвращает предыдущий локальный объект

Имя локального контекста поддерживается в том случае, если контекст был сконструирован по заданному имени или одному или нескольким именованным локальным контекстам. Если локальный контекст не имеет имени, то функция `name()` возвращает пустую строку. В этом случае стандарт также не гарантирует создания имени, полученного при объединении двух локальных контекстов. Два локальных контекста считаются

идентичными, если один из них является копией другого или если оба локальных контекста имеют одно и то же имя. Естественно считать, что два объекта идентичны, если один из них является копией другого. А что же делать с именами? Идея заключается в том, что имя локального контекста отражает имена, использованные для создания именованных фацетов. Например, имя локального контекста может быть создано с помощью объединения индивидуальных имен фацетов в определенном порядке, разделенных заданными символами. Эта схема потенциально позволяет определять идентичность двух объектов локального контекста, если они созданы путем объединения одних и тех же именованных фацетов. Иначе говоря, стандарт требует, чтобы два локальных контекста, содержащих одинаковые именованные фацеты, считались идентичными. Таким образом, имена должны создаваться тщательно, чтобы не нарушить принцип идентичности, описанный выше.

Оператор `()` позволяет использовать объект локального контекста для сравнения строк. Этот оператор сравнивает строки, переданные как аргумент, в заданном порядке с помощью операции из фацета `collate<>` (см. раздел 16.4.5). Эта операция возвращает значение `true`, если строка *str1* меньше строки *str2* по критериям локального контекста. Таким образом, локальный объект можно использовать как функциональный объект STL (см. раздел 10.1). По существу, объект локального контекста можно использовать в качестве критерия сортировки для алгоритмов STL, работающих со строками. Например, сортировку вектора по правилам сравнения строк в соответствии с немецким локальным контекстом можно реализовать следующим образом:

```
std::vector<std::string> v;
...
// сортировка строк в соответствии с немецким локальным контекстом
std::sort (v.begin(),v.end(),     // диапазон
           locale("de_DE"));      // критерий сортировки
```

16.4. Подробное описание фацетов

Фацеты — важный аспект локальных контекстов. Все локальные контексты обязательно содержат определенный набор стандартных фацетов. В описаниях отдельных фацетов, приведенных ниже, объясняется, какие специализации соответствующего фацета гарантированно содержатся в локальном контексте. Кроме перечисленных фацетов, реализация стандартной библиотеки C++ может содержать и другие фацеты. Важно помнить, что пользователь также может инсталлировать собственные фацеты или заменить ими стандартные.

В разделе 16.2.2 обсуждалась инсталляция фацета в локальном контексте. Например, класс `germanBoolNames` был объявлен производным от класса `numpunct_byname<char>`, одного из стандартных фацетов, и инсталлирован в локальном контексте с помощью конструктора, получающего локальный контекст и фацет в качестве аргументов. А что нужно для того, чтобы создать собственный фацет? В качестве фацета может использоваться любой класс `F`, удовлетворяющий двум требованиям.

1. Класс `F` является открытым наследником класса `std::locale::facet`. Этот базовый класс в основном определяет механизмы подсчета ссылок, используемые во внутренней работе объектов локального контекста. Кроме того, он делает закрытыми копирующий конструктор и оператор присваивания, предотвращая копирование или присваивание фацетов

2. Класс F содержит открытую статическую переменную id типа locale::id, которая используется для поиска фацета в объекте локального контекста по его типу. Применение типа в качестве индекса обеспечивает типовую безопасность интерфейса. Во внутреннем представлении для работы с фацетами используется обычный контейнер с целочисленным индексом.

   Стандартные фацеты соответствуют не только этим требованиям, но и двум специальным рекомендациям. Выполнение этих рекомендаций не обязательно, но полезно.

3. Все функции-члены класса объявляются константными. Это полезно, потому что функция use_facet() возвращает ссылку на константный фацет. Функции-члены, не объявленные константными, вызываться не могут.

4. Все открытые функции объявляются невиртуальными и делегируют запросы защищенной виртуальной функции. Ее имя совпадает с именем открытой функции, перед которой приписан префикс do_. Например, функция numpunct::truename() вызывает функцию numpunct::do_truename(). Этот стиль помогает избежать сокрытия функций-членов при перекрытии только одной из нескольких виртуальных функций с одинаковыми именами. Например, класс num_put содержит несколько функций с именем put(). Кроме того, при создании базового класса в невиртуальные функции можно включить дополнительный код, который будет выполняться даже в случае перекрытия виртуальных функций.

Следующее описание стандартных фацетов относится только к открытым функциям. Для того чтобы модифицировать фацет, необходимо переопределить соответствующие защищенные функции. Определение функций с таким же интерфейсом, как у открытых функций фацета, всего лишь перегрузит их, поскольку эти функции не являются виртуальными.

Для большинства стандартных фацетов определяется версия с суффиксом _byname. Она является производной от стандартного фацета и создает специализацию для соответствующего имени локального контекста. Например, класс numpunct_byname создает фацет numpunct для локального контекста с заданным именем. Скажем, немецкий фацет numpunct можно создать следующим образом:

```
std::numpunct_byname("de_DE")
```

Классы с суффиксом _byname используются конструкторами локальных контекстов, получающими имя как аргумент. Для каждого стандартного фацета, имеющего имя, создание его экземпляра осуществляется соответствующим классом с суффиксом _byname.

16.4.1. Форматирование чисел

Средства форматирования чисел преобразуют числа из внутреннего представления в соответствующее текстовое представление. Операторы потоков ввода-вывода делегируют преобразование фацетам категории locale::numeric. В эту категорию входят три фацета.

1. numpunct — пунктуация, используемая при форматировании и лексическом разборе чисел;
2. num_put — форматирование чисел;
3. num_get — лексический разбор чисел.

Короче говоря, фацет num_put выполняет форматирование чисел, описанное в разделе 15.7, посвященном потокам ввода-вывода, а num_get выполняет лексический разбор соответствующих строк. Фацет numpunct обеспечивает дополнительную гибкость, недоступную интерфейсам потоков.

Пунктуация при форматировании чисел

Фацет numpunct<> задает символ, используемый как десятичная точка, управляет вставкой необязательных разделителей между группами разрядов, а также задает текстовые представления булевых значений. Функции фацета numpunct<> приведены в табл. 16.7.

Таблица 16.7. Члены фацета numpunct<>

Выражение	Описание
np.decimal_point()	Возвращает символ, используемый как десятичная точка
np.thousands_sep()	Возвращает символ, используемый в качестве разделителя между группами разрядов
np.grouping()	Возвращает объект класса string, описывающий позиции разделителей между группами разрядов
np.truename()	Возвращает текстовое представление значения true
np.falsename()	Возвращает текстовое представление значения false

Фацет numpunct<> имеет шаблонный аргумент типа charT. Этот тип имеют символы, возвращаемые функциями decimal_point() и thousand_sep(), а функции truename() и falsename() возвращают значение типа basic_string<charT>. Специализации numpunct<char> и numpunct<wchar_t> являются обязательными.

Поскольку длинные числа трудно читать без промежуточных разделителей, стандартные фацеты числового форматирования и лексического разбора позволяют разделять группы разрядов. Чаще всего цифры, представляющие целое число, группируются по три цифры. Например, один миллион записывается так:

```
1,000,000
```

К сожалению, такое представление не является универсальным. Например, в Германии вместо запятой используется точка. Таким образом, в Германии один миллион записывается следующим образом:

```
1.000.000
```

Разделитель тысяч определяется функцией-членом thousands_sep(). Однако этого недостаточно, поскольку в некоторых странах встречаются другие правила разделения разрядов. Например, в Непале один миллион записывается в виде

```
10.00.000
```

где группы разрядов содержат даже разное количество цифр. В этой ситуации может пригодиться строка, возвращаемая функцией grouping(). Цифра в позиции с индексом *i* задает количество разрядов в *i*-й группе, причем отсчет ведется от нуля с крайней правой группы. Если количество символов в строке меньше количества групп, то размер последней

заданной группы применяется повторно. Для создания групп неограниченного размера используется значение `numeric_limits<char>::max()`, а при полном отсутствии групп используется пустая строка. В табл. 16.8 приведены примеры разного форматирования одного миллиона. Обратите внимание на то, что строка интерпретируется как последовательность целых чисел. Следовательно, обычные цифры могут оказаться неудобными (например, строка `"2"` обычно задает группу из 50 цифр, потому что символ `'2'` в кодировке ASCII имеет целочисленное значение 50).

Таблица 16.8. Примеры пунктуации при представлении одного миллиона

Значение	В виде строки	Результат
`{ 0 }`	`""`	`1000000` (по умолчанию)
`{ 3, 0 }`	`"\3"`	`1,000,000`
`{ 3, 2, 3, 0 }`	`"\3\2\3"`	`10,00,000`
`{ 2, CHAR_MAX, 0 }`	Нет	`10000,00`

Отметим, что функции `decimal_point()` и `thousands_sep()` могут возвращать `'0'`, чтобы отметить отсутствие (специальных) символов.

Форматирование чисел

Фацет `num_put<>` используется для текстового форматирования чисел. Он представляет собой шаблонный класс с двумя аргументами: тип `charT` создаваемых символов и тип `OutIt` для итератора вывода, с помощью которого записываются созданные символы. По умолчанию итератор вывода имеет тип `ostreambuf_iterator<charT>`. Фацет `num_put` содержит набор перегруженных функций, которые называются `put()` и различаются только по последнему аргументу, который задает форматируемое значение (табл. 16.9).

Таблица 16.9. Члены фацета num_put<>

Выражение	Описание
`np.put(`*to*`,`*fs*`,`*fill*`,`*val*`)`	Записывает значение *val* в итератор *to*, используя формат, заданный объектом *fs*, и символ-заполнитель *fill*

Рассмотрим аргументы функции `put()`.

- *to* — это итератор вывода, в который записывается значение. Функция `put()` возвращает копию этого итератора, установленную на позицию, расположенную сразу за последним записанным символом. Вместо него можно также передать объект потока, который будет преобразован в потоковый итератор.
- *fs* — это объект потока типа `std::ios_base`, определяющий форматирование. Обычно он представляет собой поток, содержащий требуемый локальный контекст и фацеты.
- *fill* — это символ-заполнитель.
- *val* — это форматируемое значение, которое может иметь типы `bool`, `long`, `long long`, `unsigned long`, `unsigned long long`, `double`, `long double` и `const void*`.

Использование этого фацета демонстрирует следующая программа:

```
// i18n/numput.cpp

#include <locale>
#include <chrono>
#include <ctime>
#include <iostream>
#include <exception>
#include <cstdlib>
using namespace std;

int main ()
{
   try {
       // выводим значение с плавающей точкой и
       // глобальным классическим контекстом
       locale locC;
       cout.imbue(locC);
       use_facet<num_put<char>>(locC).put (cout, cout, ' ',
                                           1234.5678);
       cout << endl;

       // выводим значение с плавающей точкой и немецким локальным контекстом
#ifdef _MSC_VER
       locale locG("deu_deu.1252");
#else
       locale locG("de_DE");
#endif
       cout.imbue(locG);
       use_facet<num_put<char>>(locG).put (cout, cout, ' ',
                                           1234.5678);
       cout << endl;
   }
   catch (const std::exception& e) {
       cerr << "Exception: " << e.what() << endl;
       return EXIT_FAILURE;
   }
}
```

Как видим, мы просто передаем поток вывода с двумя аргументами: соответствующим локальным контекстом и фацетами, предназначенными для форматирования.

```
locale locC;
cout.imbue(locC);
use_facet<num_put<char>>(locC).put (cout, cout, ' ',
                                    1234.5678);
```

На моем компьютере программа выдала следующий результат:

```
1234.57
1.234,57
```

Легко видеть, что вызов функции put() создает разные текстовые представления переданного числового значения. При работе с немецким локальным контекстом десятичная точка становится запятой, а разделитель тысяч превращается в точку, которая вставляется после каждой группы, состоящей из трех цифр.

Стандарт требует, чтобы в каждом локальном контексте хранились две специализации: num_put<char> и num_put<wchar_t>. Кроме того, стандартная библиотека C++ поддерживает все специализации, у которых первым шаблонным аргументом является символьный тип, а вторым — тип итератора вывода.

Лексический разбор чисел

Фацет num_get<> используется для лексического разбора текстовых представлений чисел. По аналогии с фацетом num_put<> он представляет собой шаблонный класс с двумя аргументами: символьным типом charT и типом итератора ввода InIt, который по умолчанию равен istreambuf_iterator<charT>. Фацет num_get<> содержит набор перегруженных функций, которые называются put() и различаются только по последнему аргументу, который задает анализируемое значение (табл. 16.10).

Таблица 16.10. Члены фацета num_get<>

Выражение	Описание
ng.get(*beg,end,fs,err,valRet*)	Выполняет лексический разбор последовательности символов в диапазоне [*beg*,*end*) в соответствии с форматом, заданным аргументом *fs*, и типом аргумента *valRet*

Этот фацет можно использовать следующим образом (полный пример содержится в файле i18n/numget.cpp):

```
std::locale     loc;              // локальный контекст
InIt            beg = ...;        // начало входной последовательности
InIt            end = ...;        // конец входной последовательности
std::ios_base&  fs = ...;         // поток, определяющий формат ввода
std::ios_base::iostate err;       // состояние после вызова
T               val;              // value after successful call

// получаем числовой фацет ввода для контекста loc
const std::num_get<charT>& ng
      = std::use_facet<std::num_get<charT,InIt>>(loc);

// считываем значение с помощью числового фацета ввода
ng.get (beg, end, fs, err, val);
```

Эти операторы выполняют попытку лексического разбора числового значения, соответствующего типу T и полученного из последовательности символов между beg и end. Формат ожидаемого числового значения определяется аргументом fs. Если разбор завершается неудачей, в переменной err устанавливается флаг ios_base::failbit. В противном случае в переменной err сохраняется флаг ios_base::goodbit, а полученное значение — в переменной value. Значение value изменяется только в случае успешного лексического разбора. Если последовательность была использована полностью, то функция get() возвращает второй параметр (end). В противном случае возвращается итератор, указывающий на первый символ, который не был проанализирован как часть числового значения.

Фацет num_get<> содержит функции для чтения объектов, имеющих типы bool, long, unsigned short, unsigned int, unsigned long, float, double, long double и void*. Некоторые типы не имеют соответствующих функций в фацете num_put, например unsigned short. Это объясняется тем, что вывод значения типа unsigned short приводит к такому же результату, что и вывод значения типа unsigned short, приведенного к типу unsigned long. Однако, если ввести значение типа unsigned long, а затем преобразовать его в тип unsigned short, может получиться число, отличающееся от результата непосредственного ввода числа типа unsigned short.

Стандарт требует, чтобы в каждом локальном контексте хранились две специализации: num_get<char> и num_get<wchar_t>. Кроме того, стандартная библиотека C++ поддерживает все специализации, у которых первым шаблонным аргументом является символьный тип, а вторым — тип итератора ввода.

16.4.2. Форматирование денежных величин

Категория monetary состоит из фацетов moneypunct, money_get и money_put. Фацет moneypunct<> определяет формат денежных величин. Два других фацета используют эту информацию для форматирования и лексического разбора денежной величины.

Пунктуация при выводе денежных величин

Вид денежной величины при выводе зависит от контекста. В разных культурных средах используются разные форматы для представления денежных величин. Например, символ денег может располагаться в разных местах (или вообще не использоваться), могут использоваться разные обозначения положительных и отрицательных величин, а также разные разделители. Для того чтобы обеспечить гибкость форматирования, подробности формата сосредоточены в фацете moneypunct<>.

Фацет moneypunct<> представляет собой шаблонный класс с двумя аргументами: типом charT и булевым значением, по умолчанию равным false. Булево значение показывает, должно ли использоваться национальное (false) или международное (true) обозначение денежных символов. Функции-члены фацета moneypunct<> приведены в табл. 16.11.

Таблица 16.11. Члены фацета moneypunct<>

Выражение	Описание
mp.decimal_point()	Возвращает символ, используемый как десятичная точка
mp.thousands_sep()	Возвращает символ, используемый как разделитель групп разрядов
mp.grouping()	Возвращает строку, определяющую расположение групп разрядов
mp.curr_symbol()	Возвращает строку с обозначением денежной единицы
mp.positive_sign()	Возвращает строку с положительным знаком
mp.negative_sign()	Возвращает строку с положительным знаком
mp.frac_digits()	Возвращает количество цифр в дробной части
mp.pos_format()	Возвращает формат, используемый для представления неотрицательных величин
mp.neg_format()	Возвращает формат, используемый для представления отрицательных величин

Следующая программа демонстрирует представления денежных величин в разных локальных контекстах:

```
// i18n/moneypunct.cpp

#include <string>
#include <iostream>
#include <locale>
#include <exception>
#include <cstdlib>
using namespace std;

// операция вывода для функций pos_format() и neg_format():
ostream& operator<< (ostream& strm, moneypunct<char>::pattern p)
{
   for (int i=0; i<4; ++i) {
        auto f = p.field[i];
        strm << (f==money_base::none ?   "none" :
                 f==money_base::space ?  "space" :
                 f==money_base::symbol ? "symbol" :
                 f==money_base::sign ?   "sign" :
                 f==money_base::value ?  "value" :
                 "???") << " ";
   }
   return strm;
}

template <bool intl>
void printMoneyPunct (const string& localeName)
{
   locale loc(localeName);
   const moneypunct<char,intl>& mp
         = use_facet<moneypunct<char,intl>>(loc);

   cout << "moneypunct in locale \"" << loc.name() << "\":" << endl;
   cout << " decimal_point: " << (mp.decimal_point()!='\0' ?
                                  mp.decimal_point() : ' ') << endl;
   cout << " thousands_sep: " << (mp.thousands_sep()!='\0' ?
                                  mp.thousands_sep() : ' ') << endl;
   cout << " grouping: ";
   for (int i=0; i<mp.grouping().size(); ++i) {
        cout << static_cast<int>(mp.grouping()[i]) << ' ';
   }
   cout << endl;
   cout << " curr_symbol: "   << mp.curr_symbol()   << endl;
   cout << " positive_sign: " << mp.positive_sign() << endl;
   cout << " negative_sign: " << mp.negative_sign() << endl;
   cout << " frac_digits: "   << mp.frac_digits()   << endl;
   cout << " pos_format: "    << mp.pos_format()    << endl;
   cout << " neg_format: "    << mp.neg_format()    << endl;
}

int main ()
```

```
{
   try {
      printMoneyPunct<false>("C");
      cout << endl;
      printMoneyPunct<false>("german");
      cout << endl;
      printMoneyPunct<true>("german");
   }
   catch (const std::exception& e) {
      cerr << "Exception: " << e.what() << endl;
      return EXIT_FAILURE;
   }
}
```

На платформе Windows программа выводит следующий результат:

```
moneypunct in locale "C":
decimal_point:
thousands_sep:
grouping:
curr_symbol:
positive_sign:
negative_sign: -
frac_digits:   0
pos_format:    symbol sign none value
neg_format:    symbol sign none value

moneypunct in locale "German_Germany.1252":
decimal_point: ,
thousands_sep: .
grouping:      3
curr_symbol:   €
positive_sign:
negative_sign: -
frac_digits:   2
pos_format:    sign value space symbol
neg_format:    sign value space symbol

moneypunct in locale "German_Germany.1252":
decimal_point: ,
thousands_sep: .
grouping:      3
curr_symbol:   EUR
positive_sign:
negative_sign: -
frac_digits:   2
pos_format:    symbol sign none value
neg_format:    symbol sign none value
```

Как видим, немецкий формат изменяет представление десятичной точки и разделителей групп разрядов (состоящих из трех цифр), а также, в зависимости от второго шаблонного параметра `intl`, символ денежной единицы — € или EUR. Отметим также, что немецкий формат отличается еще и тем, что символ евро размещается после числа, а буквы EUR — перед числом и не отделяются от него пробелом.

Подробное описание пунктуации для представления денежных величин

Фацет `moneypunct<>` является производным от класса `money_base`.

```
namespace std {
   class money_base {
      public:
         enum part { none, space, symbol, sign, value };
         struct pattern {
            char field[4];
         };
   }
};
```

Тип `pattern` предназначен для хранения четырех значений типа `part`, образующих шаблон форматирования для представления денежной единицы. В табл. 16.12 указаны пять возможных вариантов значения типа `part`, которые можно использовать в этом шаблоне.

Таблица 16.12. Части шаблонов для представления денежных единиц

Значение	Описание
`none`	В данной позиции могут находиться необязательные пробелы
`space`	В данной позиции должен находиться хотя бы один пробел
`sign`	В данной позиции может находиться знак
`symbol`	В данной позиции может находиться символ денежной единицы
`value`	В данной позиции находится значение

В фацете `moneypunct<>` определены две функции, возвращающие шаблоны форматирования: `neg_format()` для отрицательных величин и `pos_format()` для неотрицательных. В шаблоне форматирования компоненты `sign`, `symbol` и `value` являются обязательными, а `nînå` или `space` — необязательными. Однако это не означает, что на экран действительно будет выведен знак или символ денежной единицы. Содержимое соответствующих позиций зависит от значений, возвращаемых другими функциями-членами фацета, и от флагов форматирования, передаваемых функциям форматирования.

При выводе денежной величины только значение появляется в обязательном порядке. Оно выводится в позиции, отмеченной компонентом шаблона `value`. Это значение содержит точно `frac_digits()` цифр в дробной части, а в качестве десятичной точки используется символ, возвращаемый функцией `decimal_point()`. Если дробная часть отсутствует, то десятичная точка не ставится.

При вводе денежных величин разделители групп разрядов разрешены, но не обязательны. Если разделители есть, то правильность их расположения проверяется с помощью функции `grouping()`. Если функция `grouping()` возвращает пустую строку, то использование разделителей групп не разрешается. Группы разрядов разделяются символом, возвращаемым функцией `thousands_sep()`. Правила размещения разделителей между группами разрядов идентичны правилам форматирования чисел (см. раздел 16.4.1). При выводе денежных величин разделители групп разрядов всегда вставляются в соответствии со строкой, возвращаемой функцией `grouping()`. При вводе денежных величин разделители групп не обязательны, если строка группировки не является пустой. Правильность расположения разделителей проверяется после успешного завершения лексического разбора.

Отметим, что функции `decimal_point()` и `thousand_sep()` могут возвращать символ `'\0'`, сигнализируя о том, что для десятичной точки не задан специальный символ.

Компоненты `space` и `nînå` управляют размещением пробелов. Компонент `space` отмечает позицию, в которой должен находиться по крайней мере один пробел. Если при форматировании установлен флаг `ios_base::internal`, то в позиции, отмеченные компонентом `space` или `nînå`, вставляются символы-заполнители. Разумеется, заполнение производится только в том случае, если заданная минимальная ширина не заполнена другими символами. Символ-заполнитель передается как аргумент функций форматирования денежных величин. Если форматируемое значение не содержит пробелов, компонент `nînå` может находиться в последней позиции шаблона форматирования. Компоненты `space` и `nînå` не могут находиться в первой позиции шаблона форматирования, а компонент `space` не может находиться в последней позиции.

Знак денежной величины может состоять из нескольких символов. Например, в некоторых локальных контекстах отрицательные суммы заключаются в круглые скобки. В позиции компонента `sign` в шаблоне форматирования выводится первый символ представления знака. Все остальные символы выводятся в конце после всех остальных компонентов. Если строка, представляющая знак, является пустой, символы знака не выводятся. Символ, используемый для представления знака, определяется функцией `positive_sign()` для неотрицательных величин и функцией `negative_sign()` для отрицательных.

В позиции компонента `symbol` отображается символ денежной единицы. Он присутствует, только если среди флагов, используемых при форматировании или лексическом разборе, установлен флаг `ios_base::showbase`. В качестве символа денежной единицы используется строка, возвращаемая функцией `curr_symbol()`. Если второй шаблонный аргумент равен `false` (по умолчанию), то используется национальное обозначение денежной единицы; в противном случае — международное. В нашем примере, посвященном немецкому локальному контексту, использовался международный символ EUR. Если второй шаблонный аргумент фацета `moneypunct<>` равен `false`, то используется символ €.

В табл. 16.13 перечислены все варианты вывода денежной величины `$-1234.56`. Разумеется, это значит, что установлен флаг `showbase`, функция `frac_digits()` возвращает 2, а ширина равна 0.

Таблица 16.13. Примеры использования шаблонов форматирования представлений денежных величин

Шаблон	Знак	Результат
`symbol none sign value`		`$1234.56`
`symbol none sign value`	-	`$-1234.56`
`symbol space sign value`	-	`$ -1234.56`
`symbol space sign value`	()	`$ (1234.56)`
`sign symbol space value`	()	`($ 1234.56)`
`sign value space symbol`	()	`(1234.56 $)`
`symbol space value sign`	-	`$ 1234.56-`
`sign value space symbol`	-	`-1234.56 $`
`sign value none symbol`	-	`-1234.56$`

Стандарт требует, чтобы в каждом локальном контексте хранились специализации moneypunct<char>, moneypunct<wchar_t>, moneypunct<char,true> и moneypunct<wchar_t,true>.

Форматирование представлений денежных величин

Для форматирования представлений денежных величин используется фацет money_put<>. Он представляет собой шаблонный класс с двумя аргументами: первый аргумент имеет символьный тип charT, а второй — тип итератора вывода OutIt. По умолчанию итератор вывода относится к типу ostreambuf_iterator<charT>. Перегруженные функции put() создают последовательность символов в соответствии с заданным форматом (табл. 16.14).

Таблица 16.14. Члены фацета money_put<>

Выражение	Описание
tp.put(*to*, *intl*, *fs*, *fill*, *valAsDouble*)	Преобразует денежную величину, передаваемую как аргумент типа long double
tp.put(*to*, *intl*, *fs*, *fill*, *valAsString*)	Преобразует денежную величину, передаваемую как строка

Обе функции-члены put() фацета money_put<> используют следующие аргументы.

- Аргумент *to* — это итератор вывода, в который записывается денежная величина. Функция put() возвращает копию этого итератора, установленную в позицию, следующая сразу за последним записанным символом. Вместо этого аргумента можно передать поток, который будет преобразован в потоковый итератор.
- Аргумент *intl* — это булево значение, задающее используемый международный символ валюты. Таким образом, он задает второй шаблонный параметр фацета moneypunct.
- Аргумент *fs* — это объект потока типа std::ios_base, определяющий форматирование. Обычно он представляет собой поток, содержащий требуемый локальный поток, фацеты и состояние форматирования, такие как ширина поля и флаг showbase для вывода символа денежной единицы.
- Аргументы *fill* — это символ-заполнитель.
- Последний аргумент задает форматируемую денежную величину, которая может иметь тип long double или std::string. Если аргумент является строкой, она должна содержать только десятичные цифры с необязательным знаком "минус". Если первый символ строки представляет собой знак "минус", то денежная величина форматируется как неотрицательная. После того как денежная величина определена как отрицательная, знак "минус" отбрасывается. Количество цифр в дробной части определяется функцией-членом frac_digits() фацета moneypunct.

Следующая программа демонстрирует использование фацета money_put<>:

```
// i18n/moneyput.cpp

#include <locale>
#include <iostream>
```

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

int main ()
{
   try {
      // используем немецкий локальный контекст
#ifdef _MSC_VER
      locale locG("deu_deu.1252");
#else
      locale locG("de_DE");
#endif
      const money_put<char>& mpG = use_facet<money_put<char> >(locG);

      // убеждаемся, что фацет money_put<> влияет на вывод
      // и представление денежных величин
      cout.imbue(locG);
      cout << showbase;

      // для представления денежных величин используем тип double
      // (и символ из локального контекста)
      mpG.put (cout, false, cout, ' ', 12345.678);
      cout << endl;

      // для представления денежных величин используем строку
      // (и международный символ)
      mpG.put (cout, true, cout, ' ', "12345.678");
      cout << endl;
   }
   catch (const std::exception& e) {
      cerr << "EXCEPTION: " << e.what() << endl;
      return EXIT_FAILURE;
   }
}
```

Программа выводит следующий результат:

```
123,46 €
EUR123,45
```

В соответствии с форматом фацета `moneypunct` из немецкого локального контекста первый формат вывода представляет собой схему "знак значение пробел символ", использующую символ евро. Если бы использовался международный символ евро, то схема приняла бы вид "символ знак ничего значение", т.е. между символ денежной единицы и значением нет ни одного пробела.

Отметим, что в качестве денежной единицы, передаваемой функции `put()`, используется цент в США и евроцент в Европе. При передаче числа типа `long double` дробная часть денежной величины округляется. При передаче строки она просто отбрасывается.

Стандарт требует, чтобы в каждом локальном контексте содержались специализации `money_put<char>` и `money_put<wchar_t>`. Кроме того, стандартная библиотека C++ поддерживает все специализации, в которых первый аргумент имеет тип `char` или `wchar_t`, а второй является соответствующим итератором вывода.

Лексический разбор денежных единиц

Для разбора текстовых представлений денежных величин используется фацет money_get<>. Он представляет собой шаблонный класс с двумя аргументами: символьным типом charT и типом итератора ввода InIt, который по умолчанию равен istreambuf_iterator<charT>. В этом классе определены две функции-члены get(), выполняющие лексический разбор символьной последовательности, и в случае успеха сохраняющие результат в переменной типа long double или basic_string<charT> (табл. 16.15).

Таблица 16.15. Члены фацета money_get<>

Выражение	Описание
mg.get(*beg, end, intl, fs, err, valAsDoubleRet*)	Выполняет лексический разбор символьной строки [*beg,end*) в соответствии с аргументом *intl* и форматом *fs* и сохраняет результат в переменной *valAsDoubleRet* типа long double
mg.get(*beg, end, intl, fs, err, valAsStringRet*)	Выполняет лексический разбор символьной строки [*beg,end*) в соответствии с аргументом *intl* и форматом *fs* и сохраняет результат в строке *valAsStringRet*

Последовательность символов, подлежащая лексическому разбору, определяется аргументами *beg* и *end*. Разбор прекращается после обработки всех элементов или при возникновении ошибки. Если возникает ошибка, то в переменной *err* устанавливается флаг ios_base::failbit, а в аргументах valAsDoubleRet и valAsStringRet ничего не сохраняется. Если разбор закончился успешно, то результат сохраняется в переменной типа long double или basic_string<>, передаваемой по ссылке в последнем аргументе. Отметим, что денежные величины, такие как $1234.56, принимают значение 123456 как long double и"123456" как строка. В США денежной единицей является цент, а в Европе — евроцент.

Аргумент *intl* — это булево значение, являющееся индикатором выбора национального или международного обозначения денежной единицы. Фацет moneypunct<>, определяющий формат разбираемого значения, идентифицируется с помощью объекта локального контекста, связанного с аргументом fmt. При лексическом разборе денежных величин всегда используется шаблон форматирования, возвращаемый функцией neg_fîrmat() фацета moneypunct<>. В позиции none или space функция, выполняющая лексический разбор денежной величины, обрабатывает все доступные символы, если компонент none не находится в последней позиции шаблона форматирования. Завершающие пробелы не игнорируются.

Функции get() возвращают итератор, установленный в позицию, следующую сразу за последним обработанным символом

Фацет можно использовать следующим образом:

```
// получаем фацет для ввода денежных величин из локального контекста loc
const std::money_get<charT>& mg
      = std::use_facet<std::money_get<charT>>(loc);

// считываем значение с помощью фацета для ввода денежных величин
```

```
long double val;
mg.get (beg, end, intl, fs, err, val);
```

Стандарт требует, чтобы в каждом локальном контексте хранились специализации `money_get<char>` и `money_get<wchar_t>`. Кроме того, стандартная библиотека C++ поддерживает все специализации, в которых первый шаблонный аргумент является типом `char` или `wchar_t`, а второй является соответствующим типом итератором ввода.

Использование денежных манипуляторов

После принятия стандарта C++11 появилась возможность использовать манипуляторы, определенные в заголовке `<iomanip>`, для ввода денежных величин из потока и вывода их в поток. Эти манипуляторы перечислены в табл. 16.16.

Таблица 16.16. Манипуляторы для форматирования денежных величин

Манипулятор	Описание
`put_money(`*val*`)`	Записывает денежную величину *val*, используя локальный символ (вызывает функцию `put(`*strmBeg*`,`*strmEnd*`,false,`*strm*`,`*strm*`.fill(),`*val*`)` для фацета)
`put_money(`*val*`,`*intl*`)`	Записывает денежную величину *val*, используя символ, заданный аргументом *intl* (вызывает функцию `put(`*strmBeg*`,`*strmEnd*`,`*intl*`,`*strm*`,`*strm*`.fill(),`*val*`)` для фацета)
`get_money(`*valRef*`)`	Вводит денежную величину в переменную *valRef*, используя локальный символ (вызывает функцию `get(`*strmBeg*`,`*strmEnd*`,false,`*strm*`,`*err*`,`*val*`)` для фацета)
`get_money(`*valRef*`,`*intl*`)`	Вводит денежную величину в переменную *valRef*, используя символ, заданный аргументом *intl* (вызывает функцию `get(`*strmBeg*`,`*strmEnd*`,`*intl*`,`*strm*`,`*err*`,`*val*`)` для фацета)

Как и прежде, значения могут иметь тип `long double` или быть строками (или быть ссылками на эти типы), а аргумент *intl* определяет используемый символ — локальный или международный, передаваемый как второй шаблонный аргумент фацету `moneypunct`.

Следующая программа иллюстрирует работу с описанными выше манипуляторами:

```
// i18n/moneymanipulator.cpp

#include <locale>
#include <iostream>
#include <iomanip>
#include <exception>
#include <cstdlib>
using namespace std;

int main ()
{
   try {
      // используем немецкий локальный контекст
#ifdef _MSC_VER
      locale locG("deu_deu.1252");
```

```
#else
      locale locG("de_DE");
#endif

      // используем немецкий локальный контекст и       // гарантируем запись
денежной величины
      cin.imbue(locG);
      cout.imbue(locG);
      cout << showbase;

      // вводим денежную величину в переменную типа long double
      // (используя международный символ)
      long double val;
      cout << "monetary value: ";
      cin >> get_money(val,true);

      if (cin) {
          // записываем денежную величину (используя локальный символ)
          cout << put_money(val,false) << endl;
      }
      else {
          cerr << "invalid format" << endl;
      }
   }
   catch (const std::exception& e) {
      cerr << "Exception: " << e.what() << endl;
      return EXIT_FAILURE;
   }
}
```

Если ввести денежную величину

```
EUR 1234,567
```

или просто число

```
1234,567
```

то результат будет выглядеть следующим образом:

```
1.234,56 €
```

16.4.3. Форматирование времени и даты

Два фацета — `time_get<>` и `time_put<>` — из категории `time` обеспечивают поддержку лексического разбора и форматирования времени и дат. Эту задачу решают функции-члены, работающие с объектами типа `tm`. Этот тип определен в заголовочном файле `<ctime>`. Функциям передаются не сами объекты, а указатели на них.

Оба фацета в категории `time` сильно зависят от работы функции `strftime()`, также определенной в заголовочном файле `<ctime>`. Эта функция использует строки со спецификаторами преобразований и создает строку из объекта типа `tm`. Краткий обзор спецификаторов преобразований приведен в табл. 16.17. Те же самые спецификаторы использует фацет `time_put`.

Разумеется, точный вид строки, созданной функцией `strftime()`, зависит от локального контекста языка С. В таблице приведены соответствующие примеры для локального контекста `"C"`.

Форматирование времени и даты

Для форматирования времени и даты используется фацет `time_put<>`. Он представляет собой шаблонный класс с двумя аргументами: типом `charT` и типом итератора вывода `OutIt`, который является необязательным. По умолчанию итератор вывода имеет тип `ostreambuf_iterator` (см. раздел 15.13.2).

В фацете `time_put<>` определены две функции `put()`, которые преобразуют информацию о дате, хранящуюся в объекте типа `tm`, в последовательность символов, записываемую в итератор вывода. Функции фацета `time_put<>` перечислены в табл. 16.18.

Как видим, все функции-члены `put()` из фацета `time_put` используют первые четыре аргумента.

- Аргумент *to* — итератор вывода, в который записывается время. Функция `put()` возвращает копию этого итератора, установленную в позицию, следующую сразу за последним записанным символом. Вместо него можно передать поток, который будет преобразован в потоковый итератор.
- Аргумент *fs* — это объект потока типа `std::ios_base`, определяющий форматирование. Обычно он представляет собой поток, содержащий требуемый локальный контекст и фацеты.
- Аргумент *fill* — это символ заполнитель.
- Аргумент *val* — это временное значение типа `tm*`, содержащее форматируемую дату.

Первая версия функции `put()` использует аргумент *cvt* для передачи функции `strftime()` одного из спецификаторов преобразования, перечисленных в табл. 16.17 для определения форматирования.

Вторая версия функции `put()` позволяет использовать необязательный модификатор. Смысл аргумента *mod* стандартом не определен. Как показали поиски, он используется в качестве модификатора преобразований в нескольких реализациях функции `strftime()`.

Таблица 16.17. Спецификаторы преобразования функции `strftime()`

Спецификатор	Описание	Пример
`%a`	Сокращенное название дня недели	`Mon`
`%A`	Полное название дня недели	`Monday`
`%b`	Сокращенное название месяца	`Jul`
`%B`	Полное название месяца	`July`
`%c`	Представление даты и времени, заданное локальным контекстом	`Jul 12 21:53:22 1998`
`%d`	День месяца	`12`
`%H`	Час дня по 24-часовому циферблату	`21`
`%I`	Час дня по 12-часовому циферблату	`9`

Окончание табл. 16.17

Спецификатор	Описание	Пример
%j	День года	193
%m	Номер месяца	7
%M	Минуты	53
%p	Утро или вечер (AM или PM)	PM
%S	Секунды	22
%U	Номер недели, начиная с первого воскресенья	28
%W	Номер недели, начиная с первого понедельника	28
%w	Номер дня недели (воскресенье == 0)	0
%x	Представление даты, заданное локальным контекстом	Jul 12 1998
%X	Представление времени, заданное локальным контекстом	21:53:22
%y	Год без указания века	98
%Y	Год с указанием века	1998
%Z	Часовой пояс	MEST
%%	Литерал %	%

Таблица 16.18. Члены фацета `time_put<>`

Выражение	Описание
`tp.put(`*to,fs,fill,val,cvt*`)`	Преобразование в соответствии со спецификатором *cvt*
`tp.put(`*to,fs,fill,val,cvt,mod*`)`	Преобразование в соответствии со спецификатором *cvt* и модификатором *mod*
`tp.put(`*to,fs,fill,val,cbeg,cend*`)`	Преобразование в соответствии со строкой форматирования [*cbeg,cend*)

Третья версия функции `put()` использует для задания формата начало и конец последовательности символов [*cbeg,cend*), задающей требуемый формат с помощью спецификаторов преобразования функции `strftime()`. Эта версия функции `put()` управляет преобразованиями почти так же, как функция `strftime()`. Она сканирует строку и записывает каждый символ, не являющийся частью спецификации преобразования, в итератор вывода *to*. Если обнаруживается спецификатор преобразования, перед которым стоит символ %, функция извлекает необязательный модификатор и спецификатор преобразования и действует, как вторая версия функции `put()`. После этого функция `put()` продолжает сканировать строку. Отметим, что эта версия функции `put()` является довольно необычной, потому что она не вызывает соответствующую виртуальную функцию-член `do_put()`, описанную в разделе 16.4. Вместо нее она непосредственно вызывает функцию `do_put()` для второй версии. Таким образом, поведение третьей версии невозможно переопределить с помощью вывода класса из фацета `time_put`.

Следующая программа демонстрирует описанную выше информацию на примере локального контекста, заданного по умолчанию, и немецкого локального контекста:

```
// i18n/timeput.cpp

#include <locale>
#include <chrono>
#include <ctime>
#include <iostream>
#include <exception>
#include <cstdlib>
using namespace std;

int main ()
{
   try {
       // запрашиваем местное время
      auto now = chrono::system_clock::now();
      std::time_t t = chrono::system_clock::to_time_t(now);
      tm* nowTM = std::localtime(&t);

      // выводим местное время с помощью глобального классического контекста
      locale locC;
      const time_put<char>& tpC = use_facet<time_put<char>>(locC);

      // используем один спецификатор преобразования
      tpC.put (cout, cout, ' ', nowTM, 'x');
      cout << endl;

      // используем строку формата
      string format = "%A %x %I%p\n"; // формат: день_недели дата час
      tpC.put (cout, cout, ' ', nowTM,
      format.c_str(), format.c_str()+format.size() );

      // выводим местное время в соответствии с немецким локальным контекстом
#ifdef _MSC_VER
      locale locG("deu_deu.1252");
#else
      locale locG("de_DE");
#endif
      const time_put<char>& tpG = use_facet<time_put<char>>(locG);
      tpG.put (cout, cout, ' ', nowTM, 'x');
      cout << endl;
      tpG.put (cout, cout, ' ', nowTM,
               format.c_str(), format.c_str()+format.size() );
   }
   catch (const std::exception& e) {
      cerr << "Exception: " << e.what() << endl;
      return EXIT_FAILURE;
   }
}
```

Сначала мы запрашиваем местное время и преобразовываем его в структуру `struct tm*`, используя класс `std::system_clock` (см. раздел 5.7.3)[6]. Затем создаем два локальных контекста (классический контекст, заданный по умолчанию, и немецкий контекст), создаем для них фацеты `time_put` и используем их для вывода текущей даты и времени.

[6] Класс `std::system_clock` введен стандартом C++11. До принятия стандарта C++11 необходимо было объявлять объект `std::time_t t` и вызывать функцию `std::time(&t)`.

Первый вызов функции put() использует 'x' как спецификатор преобразования для функции strftime(), т.е. указывает представление даты, заданное локальным контекстом.

```
tpC.put (cout, cout, ' ', nowTM, 'x');
```

Второй вызов функции put() относится к ее третьей версии, которая использует начало и конец символьной последовательности, задающей формат вывода "*день_недели дата час*":

```
string format = "%A %x %I%p\n";
tpC.put (cout, cout, ' ', nowTM,
         format.c_str(), format.c_str()+format.size() );
```

На моем компьютере программа выдает следующий результат:

```
09/13/11
Tuesday 09/13/11 03PM
13.09.2011
Dienstag 13.09.2011 03
```

Как видно в последней строке, в Германии не используются спецификаторы AM или PM.

Стандарт требует, чтобы в каждом локальном контексте хранились две специализации — time_put<char> и time_put<wchar_t>. Кроме того, стандартная библиотека C++ поддерживает все специализации, у которых первый шаблонный аргумент является типом char или wchar_t, а второй — соответствующим типом итератора вывода.

Лексический разбор даты и времени

Фацет time_get<> представляет собой шаблонный класс, первым шаблонным аргументом которого является символьный тип charT, а вторым — тип итератора ввода InIt. По умолчанию итератор ввода имеет тип istreambuf_iterator<charT>. В табл. 16.19 перечислены функции-члены, определенные для фацета time_get<>. Все эти функции-члены, за исключением date_order(), выполняют лексический разбор строки и сохраняют результаты в объекте tm, на который ссылается аргумент t. Если лексический разбор завершился неудачей, то либо выдается сообщение об ошибке (например, с помощью модификации аргумента err), либо сохраняется неопределенное значение. Это означает, что время, вычисленное программой, обрабатывается надежно, а время, введенное пользователем, — нет. Аргумент fs задает другие фацеты, используемые при лексическом разборе. Стандарт не предусматривает других флагов, которые могут влиять на лексический разбор, кроме флага fs.

Таблица 16.19. Члены фацета time_get<>

Выражение	Описание
tg.get(*beg*, *end*, *fs*, *err*, *t*, *fmtChar*)	Выполняет лексический разбор последовательности символов [*beg*,*end*) в соответствии со спецификатором преобразований *fmtChar* (начиная со стандарта C++11)
tg.get(*beg*, *end*, *fs*, *err*, *t*, *fmtChar*, *mod*)	Выполняет лексический разбор последовательности символов [*beg*,*end*) в соответствии со спецификатором преобразований *fmtChar* и модификатором *mod* (начиная со стандарта C++11)

Окончание табл. 16.19

Выражение	Описание
`tg.get(`*beg,end,fs,err,t,fmtBeg,fmtEnd*`)`	Выполняет лексический разбор последовательности символов [*beg,end*) в соответствии с последовательностью символов [*fmtBeg,fmtEnd*) (начиная со стандарта C++11)
`tg.get_time(`*beg,end,fs,err,t*`)`	Выполняет лексический разбор последовательности символов [*beg,end*) как времени, созданного по спецификатору `X` функцией `strftime()`
`tg.get_date(`*beg,end,fs,err,t*`)`	Выполняет лексический разбор последовательности символов [*beg,end*) как даты, созданного по спецификатору `X` функцией `strftime()`
`tg.get_weekday(`*beg,end,fs,err,t*`)`	Выполняет лексический разбор последовательности символов [*beg,end*) как названия дня недели
`tg.get_monthname(`*beg,end,fs,err*`,t)`	Выполняет лексический разбор последовательности символов [*beg,end*) как названия месяца
`tg.get_year(`*beg,end,fs,err,t*`)`	Выполняет лексический разбор последовательности символов [*beg,end*) как года
`tg.date_order()`	Возвращает порядок компонентов дат, используемый фацетом

Все функции возвращают итератор, установленный в позицию, следующую сразу за последним прочитанным символом. Разбор прекращается, если строка закончилась или возникла ошибка (например, если строка не распознается как дата).

После принятия стандарта C++11 функция `get()` считывает значения в соответствии с передаваемым форматом, который либо является символом преобразования без предшествующего символа `%` и с необязательным модификатором, либо строкой форматирования, заданной началом и концом последовательности символов.

Следует отметить несколько моментов.

- Функция, считывающая название дня недели или месяца, читает как сокращенные, так и полные названия. Если за аббревиатурой следует буква, допустимая для полного названия, функция пытается прочитать полное название. Если это сделать не удается, разбор завершается неудачей, несмотря на успешный разбор сокращенного названия.
- Стандарт не регламентирует, должны ли при разборе года допускаться обозначения из двух цифр. Даже если это допускается, соответствующий год остается неопределенным.
- Функция `date_order()` возвращает порядок следования дня, месяца и года в строке даты. Это необходимо для некоторых дат, поскольку по строке, представляющей дату, невозможно определить порядок следования ее компонентов. Например, первый день февраля 2003 года может быть представлен как в виде `3/2/1`, так и в виде `1/2/3`. В классе `time_base`, базовом для фацета `time_get`, определено перечисление `dateorder` для возможных порядков следования компонентов даты. Его значения приведены в табл. 16.20.

Таблица 16.20. Члены перечисления dateorder

Значение	Описание
no_order	Порядок не определен (например, дата может быть представлена по юлианскому календарю)
dmy	День, месяц, год
mdy	Месяц, день, год
ymd	Год, месяц, день
ydm	Год, день, месяц

Следующий пример демонстрирует использование фацета time_get<>:

```
// i18n/timeget.cpp

#include <locale>
#include <ctime>
#include <iterator>
#include <iostream>
#include <string>
#include <exception>
#include <cstdlib>
using namespace std;

int main ()
{
   try {
       // используем немецкий локальный контекст
#ifdef _MSC_VER
       locale locG("deu_deu.1252");
#else
       locale locG("de_DE.ISO-8859-1");
#endif
       const time_get<char>& tgG = use_facet<time_get<char>>(locG);

      // выводим порядок компонентов даты в немецком локальном контексте
      typedef time_base TB;
      time_get<char>::dateorder d = tgG.date_order();
      cout << "dateorder: "
           << (d==TB::no_order||d==TB::mdy ? "mdy" :
               d==TB::dmy ? "dmy" :
               d==TB::ymd ? "ymd" :
               d==TB::ydm ? "ydm" : "unknown") << endl;

      // считываем день недели (в немецком формате) и время (час::мин))
      cout << "<wochentag> <hh>:<mm>: ";
      string format = "%A %H:%M";
      struct tm val;
      ios_base::iostate err = ios_base::goodbit;
      tgG.get (istreambuf_iterator<char>(cin),
               istreambuf_iterator<char>(),
               cin, err, &val,
               format.c_str(), format.c_str()+format.size());
```

```
        if (err != ios_base::goodbit) {
            cerr << "invalid format" << endl;
        }
    }
    catch (const std::exception& e) {
        cerr << "Exception: " << e.what() << endl;
        return EXIT_FAILURE;
    }
}
```

Программа может вывести следующий результат:

```
dateorder: dmy
<wochentag> <hh>:<m>:
```

Если ввести строку `Dienstag 17:30`, то все будет в порядке. Если же ввести `Tuesday 17:30` или `Dienstag 17:66`, то программа выдаст сообщение `invalid format`.

Стандарт требует, чтобы в каждом локальном контексте хранились две специализации — `time_get<char>` и `time_get<wchar_t>`. Кроме того, стандартная библиотека C++ поддерживает все манипуляторы, у которых первый шаблонный аргумент представляет собой тип `char` или `wchar_t`, а второй — соответствующий итератор ввода.

Использование манипуляторов времени

После принятия стандарта C++11 для ввода информации о времени и дате из потока или вывода ее в поток можно использовать манипуляторы, определенные в заголовочном файле `<iomanip>`. Эти манипуляторы используют функции члены `get()` или `put()` из соответствующего фацета. Манипуляторы перечислены в табл. 16.21, а пример их использования см. в разделе 15.3.3.

Таблица 16.21. Манипуляторы времени и даты

Манипулятор	Описание
`put_time` (*valPtr*, *fmt*)	Записывает информацию о времени и дате *valPtr* как объект типа `struct tm*` в соответствии с форматом *fmt*; вызывает из фацета функцию `put` (*strmBeg*, *strmEnd*, *strm*, *strm*.`fill()`, *val*, *fmtBeg*, *fmtEnd*)
`get_time` (*valPtr*, *fmt*)	Записывает информацию о времени и дате `struct tm*` *valPtr* в соответствии с форматом *fmt*; вызывает из фацета функцию `get` (*strmBeg*, *strmEnd*, *strm*, *err*, *val*, *fmtBeg*, *fmtEnd*)

16.4.4. Классификация и преобразование символов

В стандартной библиотеке C++ определены два фацета для работы с символами: `ctype<>` и `codecvt<>`. Оба фацета относятся к категории `locale::ctype`. Фацет `ctype<>` используется в основном при классификации символов, например, для проверки того, является ли символ буквой. Кроме того, он имеет методы смены регистра символов, а также преобразования между типом `char` и типом символов, для которого был специализирован данный фацет. Фацет `codecvt<>` обеспечивает переключение между кодировками символов и используется в основном шаблоном `basic_filebuf` для преобразования внутренних и внешних представлений.

Классификация символов

Фацет `ctype<>` представляет собой шаблон, параметризованный символьным типом. Класс `ctype<`*charT*`>` содержит три категории функций-членов.

1. Функции для преобразования типов `char` и *charT*.
2. Функции для классификации символов.
3. Функции для переключения между верхним и нижним регистрами.

В табл. 16.22 перечислены функции-члены, определенные для фацета `ctype`.

Таблица 16.22. Члены фацета `ctype<>`

Выражение	Описание
`ct.is(`*m*`,`*c*`)`	Проверяет, соответствует ли символ *c* маске *m*
`ct.is(`*beg*`,`*end*`,`*vec*`)`	Для каждого числа из диапазона, заданного итераторами *beg* и *end*, сохраняет маску символа в соответствующую позицию, заданную аргументом *vec*
`ct.scan_is(`*m*`,`*beg*`,`*end*`)`	Возвращает указатель на первый символ в диапазоне, заданном итераторами *beg* и *end*, соответствующий маске *m*, или *end*, если такого символа нет
`ct.scan_not(`*m*`,`*beg*`,`*end*`)`	Возвращает указатель на первый символ в диапазоне, заданном итераторами *beg* и *end*, не соответствующий маске *m* или *end*, если маске соответствуют все символы
`ct.toupper(`*c*`)`	Возвращает букву в верхнем регистре, соответствующую символу *c*; если такой буквы нет, возвращает символ *c*
`ct.toupper(`*beg*`,`*end*`)`	Преобразует каждую букву в диапазоне, заданном итераторами *beg* и *end*, заменяя ее результатом вызова функции `toupper()`
`ct.tolower(`*c*`)`	Возвращает букву в нижнем регистре, соответствующую символу *c*; если такой буквы нет, возвращает символ *c*
`ct.tolower(`*beg*`,`*end*`)`	Преобразует каждую букву в диапазоне, заданном итераторами *beg* и *end*, заменяя ее результатом вызова функции `tolower()`
`ct.widen(`*c*`)`	Возвращает символ `char` *c*, преобразованный в тип `charT`
`ct.widen(`*beg*`,`*end*`,`*dest*`)`	Для каждого символа в диапазоне, заданном итераторами *beg* и *end*, помещает результат вызова функции `widen()` в соответствующую позицию *dest*
`ct.narrow(`*c*`,`*default*`)`	Возвращает символ `charT` *c*, преобразованный в тип `char`, или символ *default*, если подходящего символа не существует
`ct.narrow(`*beg*`,`*end*`,`*default*`,`*dest*`)`	Для каждого символа в диапазоне, заданном итераторами *beg* и *end*, помещает результат вызова функции `narrow()` в соответствующую позицию *dest*

Функция `is(beg, end, vec)` сохраняет маски символов в массиве. Для каждого символа в диапазоне между итераторами *beg* и *end*, в массиве на который ссылается аргумент *vec*, сохраняется маска с атрибутами, соответствующими символам. Это позволяет избежать вызовов виртуальных функций при классификации большого количества символов.

С помощью функций `toupper()` и `tolower()` можно перевести строку в верхний или нижний регистр. Рассмотрим пример:

```
std::locale loc;
std::string s;

for (char& c : s) {
    c = std::use_facet<std::ctype<char>>(loc).toupper(c);
}
```

Функция `widen()` преобразует символ типа `char` из исходной кодировки в соответствующий символ из кодировки, используемой локальным контекстом. Следовательно, вызывать функцию `widen()` целесообразно даже в том случае, когда результат тоже относится к типу `char`. Обратное преобразование осуществляется функцией `narrow()`, которая преобразует символ из кодировки, используемой локальным контекстом, в соответствующий символ типа `char`, если такой символ существует. Например, в следующем коде строка с десятичными цифрами преобразуется из типа `char` в тип `wchar_t`:

```
std::locale loc;
char narrow[] = "0123456789";
wchar_t wide[10];

std::use_facet<std::ctype<wchar_t>>(loc).widen(narrow, narrow+10,
                                               wide);
```

Следующие вспомогательные функции преобразуют объекты типа `string` в объекты типа `wstring`, и наоборот:

```
// i18n/wstring2string.hpp

#include <locale>
#include <string>
#include <vector>

// преобразуем string в wstring
std::wstring to_wstring (const std::string& str,
                         const std::locale& loc = std::locale())
{
   std::vector<wchar_t> buf(str.size());
   std::use_facet<std::ctype<wchar_t>>(loc).widen(str.data(),
                                                  str.data()+str.size(),
                                                  buf.data());
   return std::wstring(buf.data(),buf.size());
}

// преобразуем wstring в string, используя '?' в качестве символа по умолчанию
std::string to_string (const std::wstring& str,
                       const std::locale& loc = std::locale())
{
   std::vector<char> buf(str.size());
```

```
   std::use_facet<std::ctype<wchar_t>>(loc).narrow(str.data(),
                                                   str.data()+str.size(),
                                                   '?', buf.data());
   return std::string(buf.data(),buf.size());
}
```

Эти функции можно вызвать следующим образом:

```
// i18n/wstring2string.cpp

#include <string>
#include <iostream>
#include "wstring2string.hpp"

int main()
{
   std::string s = "hello, world\n";
   std::wstring ws = to_wstring(s);
   std::wcout << ws;
   std::cout << to_string(ws);
}
```

Класс `ctype` является производным от класса `ctype_base` и используется только для определения перечисляемого типа `mask`. Это перечисление определяет значения, сочетание которых образует битовую маску для проверки свойств символов. Значения, определенные в классе `ctype_base`, приведены в табл. 16.23. Все функции классификации символов получают битовую маску, представляющую собой комбинацию значений, определенных перечислением `ctype_base`. Для создания комбинаций можно использовать побитовые операторы (`|`, `&`, `^` и `~`). Символ соответствует маске, если он совпадает с одним из символов, идентифицируемых этой маской.

Таблица 16.23. Символьные маски, используемые классом `ctype<>`

Значение	Описание
`ctype_base::alnum`	Буквы и цифры (эквивалент `alpha|digit`)
`ctype_base::alpha`	Буквы
`ctype_base::blank`	Пробел или знак табуляции (начиная со стандарта C++11)
`ctype_base::cntrl`	Управляющие символы
`ctype_base::digit`	Десятичные цифры
`ctype_base::graph`	Знаки пунктуации, буквы и цифры (эквивалент `alnum|punct`)
`ctype_base::lower`	Буквы в нижнем регистре
`ctype_base::print`	Печатные символы
`ctype_base::punct`	Знаки пунктуации
`ctype_base::space`	Пробельные символы
`ctype_base::upper`	Буквы в верхнем регистре
`ctype_base::xdigit`	Шестнадцатеричные цифры

Специализация типа ctype<> для типа char

Для того чтобы повысить производительность функций классификации символов, фацет ctype специализируется для символьного типа char. Эта специализация не делегирует функции, выполняющие классификацию символов (is(), scan() и scan_not()), соответствующим виртуальным функциям, а реализует их как подстановочные с помощью поиска по справочной таблице. Для этого в фацете определены дополнительные члены (табл. 16.24)[7].

Таблица 16.24. Дополнительные члены класса ctype<char>

Выражение	Описание
ctype<char>::table_size	Возвращает размер таблицы (>=256)
ctype<char>::classic_table()	Возвращает таблицу для "классического" локального контекста C
ctype<char>(*table*,*del*=false)	Создает фацет с таблицей *table*
ct.table()	Возвращает текущую таблицу фацета *ct*

Манипуляции с этими функциями в конкретном локальном контексте осуществляются с помощью соответствующей таблицы масок, передаваемой как аргумент конструктора.

```
// создаем и инициализируем таблицу
std::ctype_base::mask mytable[std::ctype<char>::table_size] = {
   ...
};

// используем таблицу для фацета ctype<char> ct
std::ctype<char> ct(mytable,false);
```

Этот код создает фацет ctype<char>, который определяет класс символа по таблице mytable. Точнее говоря, класс символа c определяется выражением

```
mytable[static_cast<unsigned char>(c)].
```

Статическая переменная table_size является константой, которая определяется реализацией библиотеки и хранит размер справочной таблицы. Эта таблица должна содержать не менее 256 символов. Второй необязательный аргумент конструктора ctype<char> указывает, должна ли таблица удаляться при уничтожении фацета. Если аргумент равен true, то переданная конструктору таблица освобождается с помощью оператора delete[], когда фацет становится ненужным.

Защищенная функция table() возвращает таблицу, переданную в первом аргументе конструктора. Статическая защищенная функция classic_table() возвращает таблицу, которая используется для классификации символов в классическом локальном контексте C.

Глобальные вспомогательные функции для классификации символов

Удобное использование фацета ctype<> обеспечивается стандартными глобальными функциями, перечисленными в табл. 16.25. Такую же классификацию можно провести и для регулярных выражений (см. раздел 14.8).

[7] До принятия стандарта C++11 функции ctype<char>::table() и ctype<char>::classic_table() были по ошибке определены как защищенные, а не открытые члены класса.

Таблица 16.25. Глобальные вспомогательные функции для классификации символов

Функция	Описание
`isalnum(`*c*, *loc*`)`	Проверяет, является ли символ *c* буквой или цифрой (эквивалент `isalpha()||isdigit()`)
`isalpha(`*c*, *loc*`)`	Проверяет, является ли символ *c* буквой
`isblank(`*c*, *loc*`)`	Проверяет, является ли символ *c* пробелом или знаком табуляции (начиная со стандарта C++11)
`iscntrl(`*c*, *loc*`)`	Проверяет, является ли символ *c* управляющим символом
`isdigit(`*c*, *loc*`)`	Проверяет, является ли символ *c* цифрой
`isgraph(`*c*, *loc*`)`	Проверяет, является ли символ *c* печатным, не пробельным символом (эквивалент `isalnum()||ispunct()`)
`islower(`*c*, *loc*`)`	Проверяет, является ли символ *c* буквой в нижнем регистре
`isprint(`*c*, *loc*`)`	Проверяет, является ли символ *c* печатным символом (включая пробельные символы)
`ispunct(`*c*, *loc*`)`	Проверяет, является ли символ *c* знаком пунктуации (т.е. является печатным, но не является пробелом, цифрой или буквой)
`isspace(`*c*, *loc*`)`	Проверяет, является ли символ *c* пропуском
`isupper(`*c*, *loc*`)`	Проверяет, является ли символ *c* буквой в верхнем регистре
`isxdigit(`*c*, *loc*`)`	Проверяет, является ли символ *c* шестнадцатеричной цифрой
`tolower(`*c*, *loc*`)`	Преобразует символ *c* из верхнего регистра в нижний
`toupper(`*c*, *loc*`)`	Преобразует символ *c* из нижнего регистра в верхний

Например, следующее выражение определяет, является ли символ `c` буквой в нижнем регистре в локальном контексте `loc`:

```
std::islower(c,loc)
```

Эта функция возвращает соответствующее значение типа `bool`.

Следующее выражение возвращает символ `c`, преобразованный в букву верхнего регистра в локальном контексте `loc`, если сначала символ `c` представлял собой букву в нижнем регистре:

```
std::toupper(c,loc)
```

Если символ `c` не был буквой нижнего регистра, то первый аргумент возвращается неизмененным.

Выражение

```
std::islower(c,loc)
```

эквивалентно выражению

```
std::use_facet<std::ctype<char>>(loc).is(std::ctype_base::lower,c)
```

Это выражение вызывает функция-член `is()` фацета `ctype<char>`. Функция `is()` проверяет, относится ли символ `c` к какой-либо из категорий, определяемых битовой маской в первом аргументе. Значения флагов битовой маски определены в классе `ctype_base`. Примеры использования вспомогательных функций см. в разделах 13.2.14 и 15.13.3.

Глобальные вспомогательные функции для классификации символов соответствуют одноименным функциям языка C, получающим только один аргумент. Они определены в заголовочных файлах `<cctype>` и `<ctype.h>` и всегда используют текущий глобальный контекст C.[8] Работать с ними еще удобнее:

```
if (std::isdigit(c)) {
   ...
}
```

Однако при использовании этих функций невозможно работать с разными локальными контекстами в одной и той же программе, а также вызывать функции языка C с пользовательским фацетом `ctype`. Пример применения функций C для перевода всех символов строки в верхний регистр приведен в разделе 13.2.14.

Важно отметить, что вспомогательные функции C++ не должны использоваться там, где программа должна работать максимально быстро. Получение соответствующего фацета от локального контекста и непосредственная работа с его функциями требуют намного меньше времени. Если требуется классифицировать большое количество символов в одном локальном контексте, задачу можно решить еще эффективнее, по крайней мере, для символов, не имеющих типа `char`. Для классификации типичных символов можно использовать функцию `is`(*beg*, *end*, *vec*). Для каждого символа из диапазона [*beg*,*end*) она создает маску с описанием свойств этого символа. Эта маска сохраняется в векторе *vec* на позиции, которая соответствует позиции символа в последовательности. Впоследствии этот вектор можно использовать для быстрой идентификации символов.

Преобразование кодировок символов

Фацет `codecvt<>` преобразует внутреннюю кодировку во внешнюю, и наоборот. Например, если реализация стандартной библиотеки C++ поддерживает соответствующий фацет, то с его помощью можно преобразовывать символы из кодировки Unicode в кодировку EUC (Extended UNIX Code).

Этот фацет используется классом `basic_filebuf` для выполнения преобразований между внутренней кодировкой и представлением, хранящимся в файле. Для этого класс `basic_filebuf<charT,traits>` (см. раздел 15.9.1) использует специализацию `codecvt<charT,char,typename traits::state_type>`. Применяемый фацет извлекается из локального контекста, связанного с классом `basic_filebuf`. Это основной способ использования фацета `codevect`. Непосредственно с этим фацетом работают очень редко.

Основные сведения о кодировке символов см. в разделе 16.1. Для того чтобы понять, как работает фацет `codecvt`, необходимо знать, что существуют два подхода к кодировке символов: символ может кодироваться либо фиксированным количеством байтов (широкое представление), либо переменным (многобайтовое представление).

---

[8] Этот локальный контекст идентичен глобальному объекту локального контекста C++, только если последний вызов функции `locale::global()` был выполнен для именованного локального объекта и с тех пор функция `setlocale()` не вызывалась. В противном случае локальный контекст, используемый функциями языка C, будет отличаться от глобального объекта локального контекста в языке C++.

Следует также помнить, что в многобайтовых кодировках для повышения эффективности использования памяти используется так называемое *состояние сдвига*. Для того чтобы правильно интерпретировать байт, необходимо знать правильное состояние сдвига для данной позиции. Состояние сдвига, в свою очередь, определяется только при обходе всей последовательности многобайтовых символов (см. раздел 16.1).

Фацет `codecvt<>` имеет три шаблонных аргумента.

1. Символьный тип `internT` для внутреннего представления.
2. Символьный тип `charT` для внешнего представления.
3. Тип `stateT` для представления промежуточного состояния в процессе преобразования.

Промежуточное состояние может состоять из незавершенных широких символов или текущего состояния сдвига. Стандарт C++ не ограничивает содержимое объектов, представляющих состояние.

Внутреннее представление всегда использует кодировку символов с фиксированным количеством байтов. Программы в основном работают с символьными типами `char` и `wchar_t`. Внешнее представление может использовать как фиксированную, так и многобайтовую кодировку. В случае многобайтовой кодировки второй аргумент шаблона задает тип представления ее базовых единиц. Каждый многобайтовый символ состоит из одного или нескольких объектов этого типа. Обычно для этой цели применяется тип `char`.

Третий шаблонный аргумент задает тип для представления текущего состояния преобразования. Это необходимо, например, если одна из кодировок является многобайтовой. В этом случае обработка многобайтового символа может быть прервана из-за переполнения буфера источника или получателя. Если это произошло, текущее состояние преобразования сохраняется в объекте указанного типа.

По аналогии с другими фацетами стандарт требует обязательной поддержки минимального количества специализаций. Стандартная библиотека C++ поддерживает только две специализации.

1. `codecvt<char,char,mbstate_t>` — преобразование исходной кодировки самой в себя (вырожденная версия фацета `codecvt<>`).
2. `codecvt<wchar_t,char,mbstate_t>` — преобразование между встроенной узкой кодировкой (`char`) и встроенной широкой кодировкой (`wchar_t`).

Стандарт C++ не регламентирует конкретную семантику второго преобразования. Однако единственным естественным решением было бы разделить каждый объект типа `wchar_t` на `sizeof(wchar_t)` объектов типа `char` для преобразования `wchar_t` в `char` и собрать объект типа `wchar_t` из того же количества объектов типа `char` при обратном преобразовании. Отметим, что такое преобразование радикально отличается от преобразований между типами `char` и `w_char`, которые выполняются функциями `widen()` и `narrow()` из фацета `ctype`. Функции фацета `codecvt` используют биты нескольких объектов типа `char` для создания одного объекта типа `wchar_t` (или наоборот), а функции фацета `ctype` преобразуют символ из одной кодировки в соответствующий символ другой кодировки (если он существует).

Как и фацет `ctype`, фацет `codecvt` является производным от базового класса и наследует от него перечислимый тип. Базовый класс, в котором определяется перечислимый тип `result`, называется `codecvt_base`. Значения этого перечислимого типа используются для описания результатов вызова функций-членов фацета `codecvt`. Точный смысл каждого значения зависит от вызываемой функции-члена. Функции-члены фацета `codecvt` приведены в табл. 16.26.

Таблица 16.26. Члены фацета codecvt<>

Выражение	Описание
`cvt.in(`*s*,*fb*,*fe*,*fn*,*tb*,*te*,*tn*`)`	Преобразует внешнее представление во внутреннее
`cvt.out(`*s*,*fb*,*fe*,*fn*,*tb*,*te*,*tn*`)`	Преобразует внутреннее представление во внешнее
`cvt.unshift(`*s*,*tb*,*te*,*tn*`)`	Записывает последовательность управляющих символов для переключения в исходное состояние сдвига
`cvt.encoding()`	Возвращает информацию о внешней кодировке
`cvt.always_noconv()`	Возвращает `true`, если преобразование невозможно
`cvt.length(`*s*,*fb*,*fe*,*max*`)`	Возвращает количество объектов типа `externT` в диапазоне между итераторами *fb* и *fe* для получения *max* символов во внутреннем представлении
`cvt.max_length()`	Возвращает максимальное количество объектов типа `externT`, необходимых для создания одного объекта типа `internT`

Функция `in()` преобразует внешнее представление во внутреннее. Аргумент *s* содержит ссылку на объект типа `stateT`. Перед преобразованием этот аргумент определяет начальное состояние сдвига. После выполнения преобразования в нем сохраняется финальное состояние сдвига. Переданное состояние сдвига может отличаться от начального, если текущий буфер ввода не является первым из преобразуемых буферов. Аргументы *fb* (from beginning — от начала) и *fe* (from end — от конца) имеют тип `const internT*` и определяют соответственно начало и конец буфера ввода. Аргументы *tb* (to begin — до начала) и *te* (to end — до конца) имеют тип `externT*` и определяют начало и конец буфера вывода. Аргумент *fn* (from next — от следующего) относится к типу `const extern&*&`, а аргумент *tn* (to next — до следующего) — к типу `internT*&`. Они содержат ссылки для возвращения конца преобразуемой последовательности символов в буферах ввода и вывода соответственно. Конец одного буфера может быть достигнут до того, как будет достигнут конец другого буфера. Функция возвращает значение типа `codecvt_base::result` (табл. 16.27).

Таблица 16.27. Значения, возвращаемые функциями преобразования

Значение	Описание
`ok`	Все символы источника были успешно преобразованы
`partial`	Не все символы источника были успешно преобразованы, или для создания преобразованного символа необходимы дополнительные символы
`error`	В источнике обнаружен символ, который невозможно преобразовать
`noconv`	Преобразование не требуется

Если возвращается значение `ok`, значит, функция выполнила работу без ошибок. Если `fn==fa`, то буфер ввода был обработан полностью, а результат преобразования содержится в последовательности символов в диапазоне между *tb* и *tn*. Символы этой последовательности соответствуют символам входной последовательности, возможно, с добавлением завершенного символа от предыдущего преобразования. Если аргумент *s*, переданный функции `in()`, не находился в исходном состоянии, значит, возможно, в нем хранился незавершенный символ от предыдущего преобразования.

Если функция возвращает значение `partial`, то либо выходной буфер был заполнен до завершения чтения буфера ввода, либо буфер ввода был исчерпан при наличии незавершенных символов (например, если последний байт входной последовательности был частью последовательности символов управляющей последовательности для переключения состояний сдвига). Если *fe*==*fn*, то входной буфер был исчерпан. В этом случае последовательность между итераторами *tb* и *tn* содержит все полностью преобразованные символы, а входная последовательность завершена частично преобразованным символом. Информация, необходимая для завершения преобразования символа при продолжении процесса, сохраняется в состоянии сдвига *s*. Если *fe*!=*fn*, то буфер ввода был прочитан не полностью. В этом случае *te*==*tn*, т.е. буфер вывода полон. Следующее преобразование будет продолжено с позиции *fn*.

Возвращаемое значение `noconv` является признаком особой ситуации. Оно означает, что преобразования внешнего представления во внутреннее не требовалось. В этом случае аргумент *fn* присваивается аргументу *fb*, а аргумент *tn* присваивается аргументу *tb*. В последовательности получателя не сохраняется ничего, поскольку вся необходимая информация уже хранится во входной последовательности.

Если функция возвращает значение `error`, значит, исходный символ преобразовать не удалось. Это может произойти по нескольким причинам. Например, в целевой кодировке может отсутствовать представление для соответствующего символа, или обработка входной последовательности завершилась в недопустимом состоянии сдвига. Стандарт C++ не определяет механизм, который позволил бы точнее определить причину ошибки.

Функция `out()` эквивалентна функции `in()`, но она действует в обратном направлении, преобразуя внутреннее представление во внешнее. Аргументы и возвращаемые значения двух функций совпадают; различаются только типы аргументов. Иначе говоря, аргументы *tb* и *te* имеют тип `const internT*`, а *fb* и *fe* — тип `const externT*`. Это относится и к аргументам *fn* и *tn*.

Функция `unshift()` вставляет символы, необходимые для завершения последовательности, когда в аргументе *s* передается текущее состояние преобразования. Обычно это означает, что состояние сдвига переключается в исходное. Завершается только внешнее представление. Аргументы *tb* и *tf* имеют тип `externT*`, а аргумент *tn* — тип `externT&*`. Диапазон между итераторами *tb* и *te* определяет буфер вывода, в котором сохраняются символы. Конец результирующей последовательности хранится в итераторе *tn*. Значения, возвращаемые функций `unshift()`, приведены в табл. 16.28.

Функция `encoding()` возвращает информацию о кодировке внешнего представления. Если функция `encoding()` возвращает `-1`, то преобразование зависит от состояния. Если функция `encoding()` возвращает `0`, то количество объектов типа `externT`, необходимых для построения символа во внутренней кодировке, является переменным. В противном случае функция возвращает количество объектов типа `externT`, необходимых для построения объектов типа `internT`. Эта информация может использоваться для выделения буфера соответствующего размера.

Таблица 16.28. Значения, возвращаемые функцией `unshift()`

Значение	Описание
`ok`	Последовательность завершена успешно
`partial`	Для завершения последовательности нужны дополнительные символы
`error`	Некорректное состояние
`noconv`	Для завершения последовательности символы не требуются

Функция always_noconv() возвращает значение true, если функции in() и out() никогда не выполняют преобразования. Например, стандартная реализация фацета codecvt<char, char, mbstate_t> преобразований не выполняет, поэтому для этого фацета функция always_noconv() всегда возвращает значение true. Однако это относится только к фацету codecvt локального контекста "C". Другие экземпляры этого фацета могут выполнять преобразования.

Функция length() возвращает количество объектов типа externT в диапазоне между и итераторами *fb* и *fe*, необходимых для построения *max* символов типа internT. Если диапазон между итераторами *fb* и *fe* содержит менее *max* полных символов типа internT, то возвращается количество объектов типа externT, необходимых для построения максимально возможного количества символов типа internT.

Стандартные фацеты для преобразования кода

После принятия стандарта C++11 стандартная библиотека C++ гарантирует наличие в заголовочном файле <codecvt> трех фацетов, производных от класса codecvt<> и предназначенных для преобразования кодов.

- Фацет **codecvt_utf8<>** предназначен для преобразования между многобайтовыми символьными последовательностями в кодировке UTF-8, с одной стороны, и последовательностями в кодировке UCS-2 (если в качестве символьного типа используется char16_t) или UCS-4/UTF-32 (если в качестве символьного типа используется char32_t), с другой стороны.
- Фацет **codecvt_utf16<>** предназначен для преобразования между многобайтовыми символьными последовательностями в кодировке UTF-16, с одной стороны, и последовательностями в кодировке UCS-2 (если в качестве символьного типа используется char16_t) или UCS-4/UTF-32 (если в качестве символьного типа используется char32_t), с другой стороны.
- Фацет **codecvt_utf8_utf16<>** предназначен для преобразования между многобайтовыми символьными последовательностями в кодировке UTF-8 и последовательностями в кодировке UTF-16.

Первый шаблонный параметр этих фацетов — это широкий символьный тип (char16_t, char32_t или wchar_t). Второй шаблонный параметр представляет собой максимальный код широкого символа, который может быть преобразован без выдачи сообщения об ошибке (по умолчанию 0x10FFFF). Третий шаблонный параметр — это флаг типа std::codecvt_mode, позволяющий чтение или запись маркера порядка байтов (byte order marks) или переключения в режим прямого порядка байтов (от младшего разряда к старшему).

Например, следующий фацет допускает преобразование между кодировкой UTF-8 и широкими символами типа wchar_t. Он получает маркер порядка байтов, создает маркер порядка байтов и использует для широких символов прямой порядок байтов.

```
std::codecvt_utf8<wchar_t,
                  0x10FFFF,
                  std::consume_header
                  | std::generate_header
                  | std::little_endian>
                  wchar2utf8facet;
```

Наиболее удобный способ использования этих фацетов обеспечивает класс `std::wstring_convert<>`.

Класс `wstring_convert<>`

После принятия стандарта C++11 класс `wstring_convert<>` позволяет выполнять удобные преобразования между строками широких символов и многобайтовыми строками. Перечислим его шаблонные параметры.

- Фацет для преобразования кода.
- Тип широких символов (по умолчанию `wchar_t`).
- Распределитель памяти для строки широких символов (по умолчанию `allocator<wchar_t>`).
- Распределитель памяти для многобайтовых строк (по умолчанию `allocator<char>`).

С помощью функций-членов `to_bytes()` и `from_bytes()` можно создавать многобайтовые последовательности из одного символа, строки с завершающим нулевым символом, соответствующие строке или диапазону символов, и наоборот (табл. 16.29). Функция `from_bytes()` возвращает строку, состоящую из широких символов (например, `wstring` для `wchar_t`). Функция `to_bytes()` возвращает значение типа `string`, содержащее многобайтовую последовательность символов типа `char`.

Таблица 16.29. Члены класса `wstring_convert<>`

Функция-член	Описание
`wc.from_bytes(`*c*`)`	Возвращает широкую строку, созданную из аргумента типа `char` *c*
`wc.from_bytes(`*cstr*`)`	Возвращает широкую строку, созданную из аргумента типа `const char*` *cstr*
`wc.from_bytes(`*str*`)`	Возвращает широкую строку, созданную из обычной строки *str*
`wc.from_bytes(`*cbeg*`,`*cend*`)`	Возвращает широкую строку, созданную из диапазона [*cbeg*,*cend*) символов типа `char`
`wc.to_bytes(`*c*`)`	Возвращает многобайтовую последовательность, созданную из широкого символа *c*
`wc.to_bytes(`*cstr*`)`	Возвращает многобайтовую последовательность, созданную из аргумента типа `const` *truncated** *cstr*
`wc.to_bytes(`*str*`)`	Возвращает многобайтовую последовательность, созданную из строки широких символов *str*
`wc.to_bytes(`*cbeg*`,`*cend*`)`	Возвращает многобайтовую последовательность, созданную из диапазона широких символов [*cbeg*,*cend*)

Например, следующие вспомогательные функции преобразуют объекты класса `string` в кодировке UTF-8 в объекты класса `wstring`, и наоборот:

```
// i18n/wstring2utf8.hpp

#include <codecvt>
#include <string>

// преобразуем строку в кодировке UTF-8 в объект типа wstring
std::wstring utf8_to_wstring (const std::string& str)
{
   std::wstring_convert<std::codecvt_utf8<wchar_t>> myconv;
   return myconv.from_bytes(str);
}

// преобразуем объект типа wstring в строку в кодировке UTF-8
std::string wstring_to_utf8 (const std::wstring& str)
{
   std::wstring_convert<std::codecvt_utf8<wchar_t>> myconv;
   return myconv.to_bytes(str);
}
```

Следующая программа демонстрирует применение этих функций на примере преобразования объекта типа `string` в объект типа `wstring` (с помощью функции `to_wstring()`, описанной в разделе 16.4.4) и трансформации этого объекта типа `wstring` в многобайтовую строку, которая выводится в стандартный поток вывода:

```
// i18n/wstring2utf8.cpp

#include <locale>
#include <string>
#include <iostream>
#include <exception>
#include <cstdlib>
#include "wstring2string.hpp"
#include "wstring2utf8.hpp"

int main()
{
   try {
#ifdef _MSC_VER
      // строка с немецким умлаутом и символом евро (в кодировке Windows)
      std::string s = "nj: ä + \x80 1";

      // преобразуем в строку широких символов (используя кодировку Windows
      std::wstring ws = to_wstring(s,std::locale("deu_DEU.1252"));
#else
      // строка с немецким умлаутом и символом евро (в кодировке ISO Latin-9)
      std::string s = "nj: ä + \xA4 1";
```

```
      // преобразуем строку широких символов (в кодировке ISO Latin-9)
      std::wstring ws = to_wstring(s,std::locale("de_DE.ISO-8859-15"));
#endif

      // выводим строку как последовательность в кодировке UTF-8 sequence:
      std::cout << wstring_to_utf8(ws) << std::endl;
   }
   catch (const std::exception& e) {
      std::cerr << "Exception: " << e.what() << std::endl;
      return EXIT_FAILURE;
   }
}
```

Строка `s` имеет формат восьмибитовой последовательности в кодировке Windows-1252 или ISO-8859-15, показанной на рис. 16.1, которая записывается в многобайтовую символьную последовательность в формате UTF-8.

Класс `wstring_buffer`

После принятия стандарта C++11 класс `wbuffer_convert<>` позволяет создавать потоковый буфер (см. раздел 15.13) с преобразованием широких символов в многобайтовые. Например, следующая программа преобразует последовательность символов в кодировке UTF-8 в последовательность символов в кодировке UTF-16/UCS-2:

```
// i18n/wbuffer.cpp

#include <string>
#include <iostream>
#include <codecvt>
using namespace std;

int main()
{
   // создаем поток ввода, считывающий последовательности в кодировке UTF-8
   wbuffer_convert<codecvt_utf8<wchar_t>> utf8inBuf(cin.rdbuf());
   wistream utf8in(&utf8inBuf);

  // создаем поток вывода, записывающий последовательности в кодировке UTF-16
   wbuffer_convert<codecvt_utf16<wchar_t,
                                 0xFFFF,
                                 generate_header>>
                  utf16outBuf(cout.rdbuf());
   wostream utf16out(&utf16outBuf);

   // записываем каждый прочитанный символ
   wchar_t c;
   while (utf8in.get(c)) {
       utf16out.put(c);
   }
}
```

Эта программа использует поток ввода utf8in, который считывает многобайтовые символьные последовательности в кодировке UTF-8 и преобразовывает их в широкие символы, а также поток вывода utf16out, который записывает эти символы в виде широких символов в кодировке UTF-16 и начальным маркером порядка байтов. Этот вывод почти соответствует кодировке UCS-2, потому что значения символов ограничены сверху значением 0xFFFF.

16.4.5. Сравнение строк

Фацет collate<> компенсирует различия между правилами сортировки строк. Например, в немецком языке буква *ü* при сортировке строк эквивалентна букве *u* или сочетанию *ue*. В других языках эта буква даже не считается буквой и интерпретируется как специальный символ или не интерпретируется вовсе. В этих языках действуют иные правила сортировки для определенных сочетаний символов. Для того чтобы строки сортировались в заданном пользователем порядке, можно использовать фацет collate. Функции этого фацета приведены в табл. 16.30. В этой таблице col обозначает специализацию класса collate, а в аргументах функций передаются итераторы, используемые для определения строк.

Таблица 16.30. Члены фацета collate<>

Выражение	Описание
col.compare(*beg1*, *end1*, *beg2*, *end2*)	Возвращает 1, если первая строка больше второй; 0 — если строки равны; -1 — если первая строка меньше второй
col.transform(*beg*, *end*)	Возвращает строку, предназначенную для сравнения с другими преобразованными строками
col.hash(*beg*, *end*)	Возвращает хеш-значение строки (типа long)

Фацет collate<> представляет собой шаблонный класс с аргументом, который является символьным типом charT. Строки, передаваемые функциям фацета collate, задаются с помощью итераторов типа const charT*. Это немного неудачное решение, поскольку нет гарантии, что итераторы, используемые классом basic_string<charT>, также являются указателями. Итак, сравнение строк следует проводить следующим образом:

```
std::locale loc;
std::string s1, s2;
...

// получаем фацет collate для локального контекста loc
const std::collate<char>& col = std::use_facet<std::collate<char>>(loc);

// сравниваем строки, используя фацет facet
int result = col.compare(s1.data(), s1.data()+s1.size(),
                         s2.data(), s2.data()+s2.size());
if (result == 0) {
   // строки s1 и s2 одинаковые
...
}
```

Для сравнения строк в локальном контексте существует специальная операторная функция, возвращающая результат вызова функции `compare()`.

```
bool result = loc(s1,s2); // проверяем s1<s2 по правилам контекста loc
```

Это позволяет передавать локальный контекст в качестве критерия сортировки (см. раздел 16.3).

Функция `transform()` возвращает объект класса `basic_string<charT>`. Лексикографический порядок строк, возвращаемых функцией `transform()`, совпадает с порядком следования исходных строк, сортируемых функцией `collate()`. Этот порядок позволяет ускорить работу программы, если одна строка сравнивается с большим количеством других строк. Лексикографический порядок строк в этом случае определяется гораздо быстрее, чем при использовании функции `collate()`, поскольку национальные правила сортировки могут быть относительно сложными.

Стандартная библиотека C++ требует обязательной поддержки только двух специализаций — `collate<char>` и `collate<wchar_t>`. Для других типов символов пользователи должны писать свои собственные специализации, возможно, используя стандартные специализации.

16.4.6. Интернационализация сообщений

Фацет `messages<>` используется для извлечения интернационализированных сообщений из каталога. Как правило, он используется для предоставления сервиса, аналогичного функции `perror()`. В системах POSIX эта функция выводит системное сообщение об ошибке, номер которой хранится в глобальной переменной `errno`. Фацет `messages<>` является гораздо более гибким. К сожалению, в стандарте он определен недостаточно четко.

Фацет `messages<>` представляет собой шаблонный класс с аргументом, являющимся символьным типом `charT`. Строки, возвращаемые этим фацетом, имеют тип `basic_string<charT>`. Этот класс открывает каталог, читает сообщения и закрывает каталог. Класс `messages` является производным от класса `messages_base`, в котором определяется тип `catalog`, представляющий собой определение типа для `int`. Объект типа `catalog` идентифицирует каталог, с которым работают функции фацета `messages`. Список этих функций приведен в табл. 16.31.

Таблица 16.31. Члены фацета `messages<>`

Выражение	Описание
`msg.open(`*name*`,`*loc*`)`	Открывает каталог и возвращает соответствующий идентификатор
`msg.get(`*cat*`,`*set*`,`*msgid*`,`*def*`)`	Возвращает сообщение с идентификатором *msgid* из каталога *cat*; если такое сообщение отсутствует, возвращает *def*
`msg.close(`*cat*`)`	Закрывает каталог *cat*

Имя, передаваемое как аргумент функции `open()`, идентифицирует каталог, в котором хранятся строки сообщений. Например, это может быть имя файла. В аргументе *loc* передается объект `locale`, используемый для обращения к фацету `ctype`. Этот фацет обеспечивает преобразование сообщений в требуемый тип символов.

Точная семантика функции `get()` не определена. Например, реализация для систем POSIX может вернуть строку, соответствующую сообщению об ошибке *msgid*, но такое

поведение не требуется стандартом. Аргумент *set* предназначен для дополнительного структурирования сообщений. Например, с его помощью можно различать системные ошибки и ошибки стандартной библиотеки C++.

Когда каталог больше не нужен, он освобождается функцией `close()`. Несмотря на то что интерфейс `open()` и `close()` предполагает, что сообщения читаются из файла по мере необходимости, такое поведение не является обязательным. Например, более вероятно, что функция `open()` прочитает файл и сохранит сообщения в памяти. Последующий вызов функции `close()` освободит эту память.

Стандарт требует, чтобы в каждом локальном контексте хранились две специализации — `messages<char>` и `messages<wchar_t>`. Другие специализации стандартной библиотекой C++ не поддерживаются.

Глава 17

Работа с числами

В главе описываются компоненты стандартной библиотеки C++, предназначенные для работы с числами, в частности, компоненты для работы со случайными числами и распределениями, классы комплексных чисел, глобальные численные функции, унаследованные от библиотеки C, и массивы значений.

Но кроме указанных выше инструментов стандартная библиотека C++ содержит и другие компоненты для работы с числами.

1. Для всех основных числовых типов определены свойства `numeric_limits`, зависящие от реализации (см. раздел 5.3).
2. Класс `ratio<>` поддерживает дробную арифметику, главным образом в качестве основы для работы со временем (см. раздел 5.6).
3. Библиотека STL содержит некоторые числовые алгоритмы, описанные в разделе 11.11.
4. Стандартная библиотека C++ содержит класс `valarray` для работы с массивами чисел. Однако на практике этот класс почти не используется, поэтому он лишь кратко упоминается в разделе 17.4, а его полное описание приводится в приложении, доступном по адресу `http://www.cppstdlib.com`.

17.1. Случайные числа и распределения

После принятия стандарта C++11 в стандартную библиотеку C++ включена библиотека для работы со случайными числами и распределениями, состоящая из большого набора классов и типов, способных удовлетворить потребности как новичков, так и экспертов. Она более сложная, чем может ожидать наивный программист. Разумеется, она содержит многие хорошо известные *распределения*. Но помимо этого стандартная библиотека C++ содержит много *генераторов случайных чисел* (engines). Эти генераторы порождают случайные *значения* без знака, равномерно распределенные между заданными минимумом и максимумом, а распределения преобразуют эти значения в случайные *числа*, линейно или нелинейно распределенные в соответствии с параметрами, заданными пользователем[1]. Таким образом, значения, порожденные генераторами, не следует использовать непосредственно (подробности изложены в разделе 17.1.1). Библиотека содержит много генераторов, потому что никакие числа в компьютере не являются истинно случайными и приходится прилагать много усилий, для того чтобы добиться высокой степени случайности, так что генераторы отличаются друг от друга по качеству, размеру и скорости работы.

[1] С формальной точки зрения результат работы функции распределения правильнее называть *псевдослучайными*, а не *случайными числами*.

Отметим, что термин *генератор псевдослучайных чисел* является довольно двусмысленным, потому что он применяется к двум сущностям.

1. Он может относиться к источнику случайности. В соответствии со стандартом каждый генератор должен соответствовать требованиям, предъявляемым к генератору равномерно распределенных случайных чисел.
2. Он может относиться к механизму для генерации случайных чисел, представляющему собой генератор равномерно распределенных чисел и распределение.

Говоря о генераторе случайных чисел в программе, мы чаще всего будем иметь в виду второе утверждение. Таким образом, для работы нам необходимы генератор и распределение, что, как показано в первом примере, не вызывает затруднений.

Для использования библиотеки случайных чисел в программу следует включить заголовочный файл `<random>`.

17.1.1. Первый пример

Прежде чем перейти к обсуждению деталей, рассмотрим первый пример, в котором случайные числа используются для перетасовки элементов. Качество случайных чисел и перетасовки в данном случае не имеет значения.

```
// num/random1.cpp

#include <random>
#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
   // создаем генератор случайных чисел по умолчанию
   std::default_random_engine dre;

   // используем генератор для порождения целых чисел
   // в интервале от 10 до 20 (включая оба эти числа)
   std::uniform_int_distribution<int> di(10,20);
   for (int i=0; i<20; ++i) {
        std::cout << di(dre) << " ";
   }
   std::cout << std::endl;

   // используем генератор для порождения чисел с плавающей точкой
   // в интервале от 10.0 до 20.0 (включая 10.0 и исключая 20.0)
   std::uniform_real_distribution<double> dr(10,20);
   for (int i=0; i<8; ++i) {
        std::cout << dr(dre) << " ";
   }
   std::cout << std::endl;

   // используем генератор для перетасовки элементов
   std::vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
   ...
```

```
    std::shuffle (v.begin(), v.end(),  // интервал
                  dre);                // источник случайности
    for (int i=0; i<v.size(); ++i) {
         std::cout << v[i] << " ";
    }
    std::cout << std::endl;
}
```

Эта программа демонстрирует общий подход к библиотеке случайных чисел. После включения заголовочного файла для библиотеки случайных чисел

```
#include <random>
```

мы создаем случайные числа, комбинируя *генератор* `dre` с двумя *распределениями* `di` и `dr`.

- **Генераторы случайных чисел** являются источником случайности, имеющим состояние (stateful source of randomness). Это функциональные объекты, способные генерировать равномерно распределенные случайные *значения* без знака в интервале, заданном минимальным и максимальным значениями.
- **Распределения** позволяют указать, как на основе этих случайных значений создать случайные *числа*, распределенные на заданном интервале в соответствии с параметрами, заданными пользователем.

Стандартная библиотека C++ содержит много генераторов случайных чисел, потому что существует много подходов к их созданию (алгоритмы и их реализации различаются как по качеству, так и по быстродействию). В данном случае мы используем генератор, заданный по умолчанию, поскольку наша задача "обычная, простая и/или не требует особой точности".

```
std::default_random_engine dre;
```

В соответствии со стандартом такой генератор можно назвать "приемлемым ...с точки зрения быстродействия, размера, качества или сочетания этих факторов". Тем не менее выбор генератора по умолчанию зависит от реализации. По этой причине результаты работы программы на разных платформах могут оказаться разными.

Стандартная библиотека C++ содержит много распределений для разных типов числовых данных: равномерное, нормальное/гауссовское, экспоненциальное, гамма, Бернулли и др. Наиболее часто используемым является равномерное распределение, которое предполагает, что числа равномерно распределены по интервалу, заданному минимальным и максимальным значениями. Для этого распределения предусмотрены два класса: один для целых чисел, другой для действительных.

Сначала мы используем класс `uniform_int_distribution`, генерирующий целые числа. Эти числа могут иметь тип `short`, `int`, `long`, `long long` и соответствующие им типы без знаков. Если тип не указан, по умолчанию используется тип `int`. Первый аргумент конструктора задает минимальное значение (по умолчанию: `0`), а второй — максимальное значение (по умолчанию: `numeric_limits<`*тип*`>::max()`; см. раздел 5.3). Отметим, что минимальное и максимальное числа можно генерировать, поэтому интервал *не является* полуоткрытым. Следовательно, следующий оператор определяет, что случайные числа, созданные генератором `di`, распределены между значениями `10` и `20` (включая оба):

```
std::uniform_int_distribution<int> di(10,20);
```

Для генерации чисел в классе распределения предусмотрена операторная функция `()`, получающая генератор как аргумент. Таким образом, для создания следующего случайного числа, соответствующего этому распределению, следует вызвать конструктор

```
di(dre)
```

Отметим, что начальное состояние генератора точно задано и не является случайным. Таким образом, следующий код дважды выведет одно и то же значение:

```
std::uniform_int_distribution<int> d;
std::default_random_engine dre1;
std::cout << d(dre1) << " ";
std::default_random_engine dre2;
std::cout << d(dre2) << " ";
```

Если требуется непредсказуемое случайное число, необходимо случайным образом задать состояние генератора с помощью параметра, на который программа не может влиять, например, количество миллисекунд между двумя щелчками мышью. Иначе говоря, необходимо задать *начальное значение* (seed) для конструктора генератора. Рассмотрим пример:

```
unsigned int seed = ...                 // задаем действительно случайное значение
std::default_random_engine dre(seed); // и используем его как начальное
                                        // состояние генератора
```

Кроме того, с помощью *начального значения* можно изменить состояние уже существующего генератора, используя функцию-член.

Точно так же генерируются числа с плавающей точкой. Однако в этом случае минимум и максимум действительно определяют полуоткрытый интервал. По умолчанию интервал имеет вид [0.0,1.0). Возможны следующие типы: `float`, `double` или `long double`, причем `double` — тип, используемый по умолчанию. Таким образом, следующий оператор определяет распределение `dr` и генерирует значение в интервале от `10.0` до `20.0` (исключая `20.0`).

```
std::uniform_real_distribution<double> dr(10,20);
```

В противоположность этому следующая инструкция заставляет объект `d` генерировать значения типа `double` в интервале от `0.0` до `0.9999...`, т.е. в интервале, не содержащем число `1.0`:

```
std::uniform_real_distribution<> d;
```

В заключение программа демонстрирует, как с помощью генератора случайных чисел перетасовать элементы в контейнере или интервале. Начиная со стандарта C++11 в библиотеку включен алгоритм `shuffle()` (см. раздел 11.8.4), использующий генератор равномерно распределенных случайных чисел, такой как `std::default_random_engine`, для перетасовки элементов.

```
std::default_random_engine dre;
...
std::shuffle (v.begin(), v.end(), // интервал
              dre);               // источник случайности
```

Результат работы программы может выглядеть следующим образом[2]:

```
10 11 18 15 15 12 10 17 17 20 14 15 19 10 10 15 17 10 14 10
16.8677 19.3044 15.2693 16.5392 17.0119 17.622 10.4746 13.2823
1 6 3 4 2 8 9 5 7
```

Будьте осторожны, работая с временными генераторами

Программист может, но не должен передавать в качестве аргумента только что созданный временный генератор[3]. Причина заключается в том, что при каждой инициализации генератора его начальное состояние оказывается одним и тем же. Таким образом, если программа содержит инструкции, приведенные ниже, то дважды будет выполнена одна и та же перетасовка.

```
std::shuffle(v.begin(),v.end(),                  // интервал
             std::default_random_engine()); // генератор случайных чисел
...
std::shuffle(v.begin(),v.end(),                  // интервал
             std::default_random_engine()); // генератор случайных чисел
```

Другими словами, если первый элемент в первый раз был переставлен в конец интервала, то в следующий раз первый элемент снова будет переставлен в конец интервала. Таким образом, при каждом вызове перетасовка модифицирует позиции каждого элемента одним и тем же способом. Рассмотрим пример:

```
1 2 3 4 5 6 7 8 9  // начальное состояние
8 7 5 6 2 4 9 3 1  // после первой перетасовки
3 9 2 4 7 6 1 5 8  // после второй перетасовки с помощью
                   // генератора с неизменным состоянием
```

Этот пример показывает, что перетасовка может создать проблемы, не говоря уже о том, что ее результат после первого вызова становится предсказуемым. Независимо от того, насколько часто вызывается алгоритм `shuffle()`, число `4` всегда будет занимать только четвертую или шестую позицию соответственно текущему состоянию, а алгоритм `shuffle()` просто переставляет местами эти две позиции.

Для того чтобы избежать этой ловушки, следует использовать следующий код, в котором перетасовка каждый раз производится с новым состоянием:

```
std::default_random_engine dre;
...
std::shuffle(v.begin(),v.end(), // интервал
             dre);              // генератор случайных чисел
...
std::shuffle(v.begin(),v.end(), // интервал
             dre);              // генератор случайных чисел
```

---

[2] Здесь использовано слово "может", потому что точное определение генератора `default_random_engine` зависит от реализации. В данном случае в качестве генератора `default_random_engine` используется класс `minstd_rand0`.

[3] Благодарю за это замечание Уолтера Э. Брауна (Walter E. Brown).

Теперь каждая перетасовка отличается от другой, и результат может выглядеть следующим образом:

```
1 2 3 4 5 6 7 8 9  // начальное состояние
8 7 5 6 2 4 9 3 1  // после первой перетасовки
3 6 2 7 1 5 8 9 4  // после второй перетасовки с помощью генератора,
                   // имеющего разные состояния
```

Не используйте генераторы без распределений

Наивный программист может спросить: почему просто не использовать значения, созданные генератором, в качестве случайных чисел, а если интервал не совпадет с требуемым, просто использовать деление по модулю `%`?

Ответ на этот важный вопрос дан в разделе 7.4.4 книги Andrew Koenig и Barbara E. Moo *Accelerated C++* (см. [*KoenigMoo:Accelerated*])[4]. Авторы этой книги объясняют, почему использование стандартного генератора случайных чисел `rand()` из языка C порождает проблемы. Существо дела состоит в следующем (точные цитаты выделены *курсивом*).

Вычисление случайных чисел с помощью операции `rand()%n` *на практике является ошибочным по двум причинам.*

1. Если *n* — небольшое целое число, то многие реализации генераторов случайных чисел выдают остатки, которые не являются в достаточной степени случайными. Например, довольно часто последовательные результаты функции `rand()` состоят из чередующихся четных и нечетных чисел. В этом случае, если `n` равно 2, последовательные результаты операции `rand()%n` будут принимать чередующиеся значения 0 и 1.
2. С другой стороны, *если число n велико, а максимальное генерируемое значение не делится на* `n` *нацело, то некоторые остатки будут появляться чаще других.* Например, если максимум равен `32767`, а `n` равно `2000`, то `17` сгенерированных значений (`500`, `2500`, ..., `30500`, `32500`) будут преобразованы в `500` и только `16` сгенерированных значений (`1500`, `3500`, ..., `31500`) будут отображены в `1500`. Причем, чем больше `n`, тем хуже становится ситуация.

Хорошие линейные генераторы учитывают эти ситуации при отображении сгенерированных значений в целевой интервал случайных чисел. Таким образом, для того чтобы получить случайные числа хорошего качества, генераторы и распределения всегда следует использовать совместно.

17.1.2. Генераторы

Стандартная библиотека C++ содержит 16 генераторов случайных чисел, которые можно использовать для вычисления случайных чисел, комбинируя их с распределениями или перетасовками (рис. 17.1).

Как указывалось выше, генератор случайных чисел — это *источник случайности, имеющий состояние*. Это состояние определяет последовательность генерируемых случайных *значений*, но не случайных *чисел* (см. раздел 17.1.1). При каждом вызове функции с помощью

[4] Русский перевод: Кениг Э., Му Б. *Эффективное программирование на C++*. — М.: Вильямс, 2002. — 382 с. — *Примеч. ред.*

оператора () возникает новое случайное значение без знака, а внутреннее состояние генератора изменяется, чтобы в следующий раз появилось новое случайное значение.

Переходы между состояниями и генерируемые значения обычно точно определены. Таким образом, при одном и том же состоянии конкретный генератор на любой платформе создаст одно и то же случайное значение. Единственным исключением является генератор default_random_engine, зависящий от реализации. Он вычисляет предсказуемые значения, но алгоритмы, лежащие в его основе, на разных платформах могут быть разными.

Можно, конечно, сказать, что каждое значение, созданное генератором, всегда должно быть случайным, но на компьютере ни одно число не бывает истинно случайным. Если требуется действительно непредсказуемое значение, состояние генератора следует обрабатывать случайным образом, не зависящим от программы, например, определяя количество миллисекунд между двумя последовательными щелчками мышью.

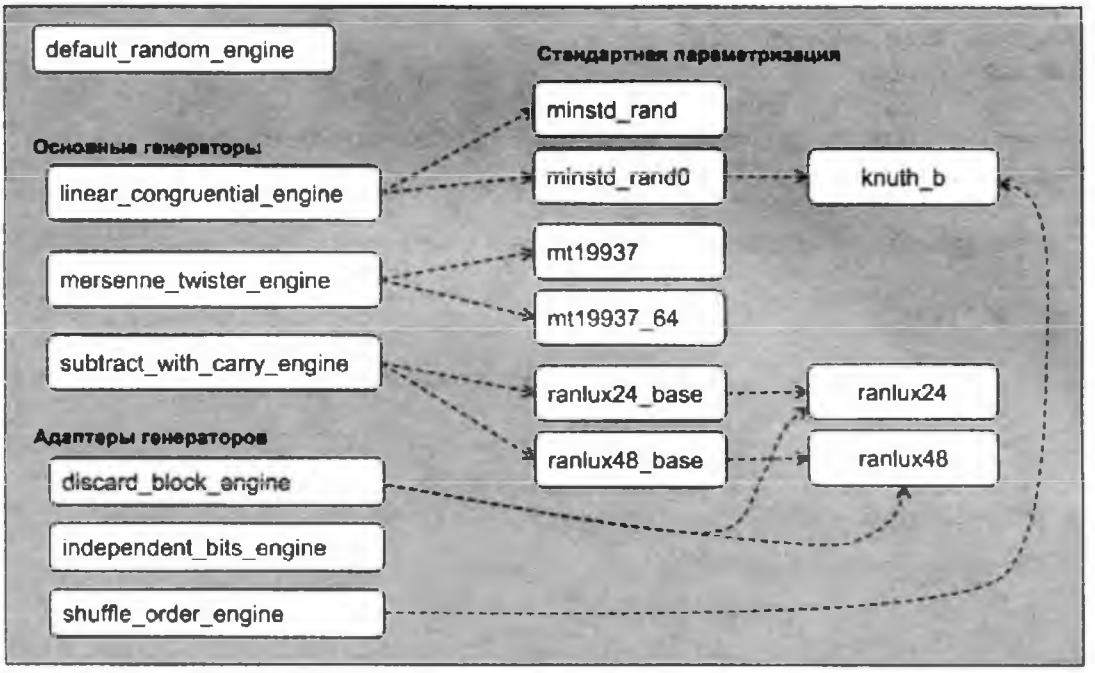


Рис. 17.1. Стандартные генераторы псевдослучайных чисел

С другой стороны, предсказуемость случайных величин дает преимущество: можно создавать одинаковые "случайные" сценарии для тестирования на основе случайных чисел.

Эту возможность демонстрирует следующая программа:

```
// num/random2.cpp

#include <random>
#include <iostream>
#include <sstream>

void printNumbers (std::default_random_engine& dre)
{
   for (int i=0; i<6; ++i) {
        std::cout << dre() << " ";
   }
   std::cout << std::endl;
}
```

```
int main()
{
   // создаем генератор и генерируем случайные числа
   std::default_random_engine dre;
   printNumbers(dre);

   // создаем аналогичный генератор и генерируем числа
   std::default_random_engine dre2; // начальное состояние, как у генератора dre
   printNumbers(dre2);

   // создаем генератор с начальным состоянием, заданным начальным значением
   std::default_random_engine dre3(42);
   printNumbers(dre3);

   // сохранить состояние генератора dre
   std::stringstream engineState;
   engineState << dre;

   // генерируем числа в соответствии с текущим состоянием генератора dre
   printNumbers(dre);

   // восстанавливаем сохраненное состояние генератора dre и снова создаем числа
   engineState >> dre;
   printNumbers(dre);

   // восстанавливаем сохраненное состояние генератора dre, пропускаем 2 числа
   // и снова создаем числа
   engineState.clear();   // создаем бит EOF
   engineState.seekg(0);  // и перемещаем позицию чтения в начало
   engineState >> dre;
   dre.discard(2);
   printNumbers(dre);
}
```

Программа может вывести следующий результат[5]:

```
16807 282475249 1622650073 984943658 1144108930 470211272
16807 282475249 1622650073 984943658 1144108930 470211272
705894 1126542223 1579310009 565444343 807934826 421520601
101027544 1457850878 1458777923 2007237709 823564440 1115438165
101027544 1457850878 1458777923 2007237709 823564440 1115438165
1458777923 2007237709 823564440 1115438165 1784484492 74243042
```

После объявления генератора случайных чисел (в нашем случае — заданного по умолчанию) для создания случайных значений можно выполнить оператор `()`.

```
std::default_random_engine dre;
...
dre()
```

Таким образом, генератор — это функциональный объект (т.е. объект, который ведет себя как функция; см. раздел 6.10).

[5] Точный вывод будет другим, потому что генератор `default_random_engine` зависит от реализации. Здесь значения соответствуют определению генератора `default_random_engine` в виде `minstd_rand0`.

Генерируются целые значения без знака. Обычно можно задать точный тип, за исключением `default_random_engine`, который зависит от реализации. Для каждого генератора операция `result_type` определяет тип, а статические функции-члены `min()` и `max()` возвращают минимальные и максимальные генерируемые значения (в интервал включаются оба).

Если снова объявить такой же генератор, он будет иметь такое же начальное состояние и будет создавать те же самые случайные значения. Для того чтобы получить то же самое значение, можно дважды передать генератор функции `printNumbers()` по значению. Поскольку функция `printNumbers()` при каждом вызове будет работать с временной копией генератора `dre`, будут дважды созданы одни и те же случайные числа.

Следующий оператор показывает, как передать начальное значение, чтобы изменить начальное состояние генератора:

```
std::default_random_engine dre3(42);
```

Обычно, чтобы получить действительно случайные числа, в качестве начального значения необходимо передать значение, поступающее извне программы (например, интервал времени между двумя щелчками мышью или значение, выданное таймером).

Следующий код демонстрирует другие возможности генераторов. Например, для сохранения и восстановления состояния генератора можно использовать операторы `<<` и `>>`.[6] Следующие инструкции записывают состояние в строку с помощью строкового потока (см. раздел 15.10):

```
std::ostringstream engineState;
engineState << dre;
```

Это состояние представляет собой последовательность значений, разделенных пробелами. (Но она *не является* последовательностью случайных чисел!) Если изменить состояние генератора `dre` — например, генерируя числа, — а затем восстановить его, то генератор перейдет в состояние, которое было сохранено, и сгенерирует те же числа снова.

Следующий оператор пропускает два состояния:

```
dre.discard(2);
```

Таким образом, генератор `dre` будет находиться в таком состоянии, будто он уже сгенерировал два значения. Однако зачастую функция `discard()` при переборе состояний с помощью оператора `()` может работать быстрее.

17.1.3. Подробное описание генераторов

Как показано на рис. 17.1 (раздел 17.1), генераторы в стандартной библиотеке шаблонов разделяются на несколько категорий.

- **Основные генераторы**, реализующие основные алгоритмы генерации случайных значений.
  - Класс `std::`**linear_congruential_engine**
  - Класс `std::`**mersenne_twister_engine**
  - Класс `std::`**subtract_with_carry_engine**

[6] Вероятно, это первый и единственный интерфейс сериализации в стандартной библиотеке C++.

- **Адаптеры генераторов**, которые можно инициализировать (основными) генераторами.
  - Класс `std::`**discard_block_engine**, адаптирующий генератор, каждый раз отбрасывает заданное количество сгенерированных значений.
  - Класс `std::`**independent_bits_engine**, адаптирующий генератор для создания случайных значений с заданным количеством битов.
  - Класс `std::`**shuffle_order_engine**, адаптирующий генератор с помощью изменения порядка генерируемых им значений.
- **Адаптеры с предопределенными параметрами.**
  - `std::`**minstd_rand0**
  - `std::`**minstd_rand**
  - `std::`**mt19937**
  - `std::`**mt19937_64**
  - `std::`**ranlux24_base**
  - `std::`**ranlux48_base**
  - `std::`**ranlux24**
  - `std::`**ranlux48**
  - `std::`**knuth_b**

Например, тип `knuth_b` — это сокращение для выражения

```
shuffle_order_engine<linear_congruential_engine<uint_fast32_t,
                                                16807,
                                                0,
                                                2147483647>,
                     256>
```

Кроме того, по умолчанию задается тип `std::default_random_engine`, зависящий от реализации. Это единственный тип генератора, не гарантирующий создания одной и той же последовательности на разных платформах.

В табл. 17.1 перечислены операции над генераторами случайных чисел. Все стандартные генераторы одного и того же типа изначально имеют одно и то же состояние, т.е. оператор == возвращает значение `true` и оба генератора создают одну и ту же последовательность случайных значений. Таким образом, для создания разных случайных значений необходимо задавать разные параметры *seed* .

Таблица 17.1. Операции над генераторами случайных чисел

Операция	Описание
engine e	Конструктор по умолчанию; создает генератор с начальным состоянием, заданным по умолчанию
engine e (*seed*)	Создает генератор, состояние которого соответствует параметру *seed*
engine e (*e2*)	Копирующий конструктор; копирует генератор (*e* и *e2* будут иметь одинаковые состояния)

Окончание табл. 17.1

Операция	Описание
`e.seed()`	Переводит генератор `e` в начальное состояние
`e.seed(`*seed*`)`	Переводит генератор `e` в состояние, соответствующее параметру *seed*
`e()`	Возвращает следующее случайное значение и переводит генератор в следующее состояние
`e.discard(`*n*`)`	Переводит генератор в *n*-е состояние (эквивалентно *n* вызовам `e()`, но может работать быстрее)
`e1 == e2`	Проверяет равенство состояний генераторов `e1` и `e2`
`e1 != e2`	Проверяет неравенство состояний генераторов `e1` и `e2`
ostrm `<< e`	Записывает состояние генератора `e` в поток вывода `ostrm`
istrm `>> e`	Считывает новое состояние из потока ввода в генератор `e`

Состояние, считанное оператором <<, представляет собой список десятичных значений, разделенных пробелом. Несмотря на то что этот список может выглядеть как последовательность случайных значений, на самом деле *это не так.*

Стандарт гарантирует, что при чтении записанного состояния генератор того же типа перейдет в то же состояние (оператор == возвращает `true`, и генерируются одинаковые случайные значения).

17.1.4. Распределения

Как было указано выше, распределения преобразуют случайные *значения*, созданные генератором, в реальные и полезные случайные *числа*. Вероятность этих случайных чисел зависит от используемого распределения, параметры которого задаются программистом. В табл. 17.2 приведен обзор всех распределений, существующих в стандартной библиотеке C++.

Таблица 17.2. Распределения, предусмотренные стандартной библиотекой C++

Категория	Имя	Тип данных
Равномерные распределения	`uniform_int_distribution`	*IntType*
	`uniform_real_distribution`	*RealType*
Распределения Бернулли	`bernoulli_distribution`	`bool`
	`binomial_distribution`	*IntType*
	`geometric_distribution`	*IntType*
	`negative_binomial_distribution`	*IntType*
Распределения Пуассона	`poisson_distribution`	*IntType*
	`exponential_distribution`	*RealType*
	`gamma_distribution`	*RealType*
	`weibull_distribution`	*RealType*
	`extreme_value_distribution`	*RealType*

Окончание табл. 17.2

Категория	Имя	Тип данных
Нормальные распределения	`normal_distribution`	*RealType*
	`lognormal_distribution`	*RealType*
	`chi_squared_distribution`	*RealType*
	`cauchy_distribution`	*RealType*
	`fisher_f_distribution`	*RealType*
	`student_t_distribution`	*RealType*
Выборочные распределения	`discrete_distribution`	*IntType*
	`piecewise_constant_distribution`	*RealType*
	`piecewise_linear_distribution`	*RealType*

Почти все распределения представляют собой шаблонные классы, параметризованные типами генерируемых значений. Единственным исключением является класс `bernoulli_distribution`, представляющий собой обычный класс, потому что он генерирует только значения типа `bool`.

По умолчанию используются следующие типы:

- `int` — для *IntType*;
- `double` — для *RealType*;

Стандартная библиотека C++ содержит следующие специализации шаблонов:

- для *IntType* — `short`, `int`, `long`, `long long` и соответствующие им типы без знаков;
- для *RealType* — `float`, `double`, `long double`.

В табл. 17.3 приведены операции над распределениями.

Таблица 17.3. Типы и операции над распределениями

Операция	Описание
distr`::result_type`	Арифметический тип генерируемых значений
distr `d`	Конструктор по умолчанию; создает распределение с параметрами, заданными по умолчанию
distr `d(`*args*`)`	Создает распределение, параметризованное списком аргументов *args*
`d(`*e*`)`	Возвращает следующее значение с помощью генератора *e* и переводит генератор `d` в состояние генератора *e*
`d.min()`	Возвращает минимальное значение
`d.max()`	Возвращает максимальное значение
`d1 == d2`	Проверяет равенство состояний генераторов `d1` и `d2`
`d1 != d2`	Проверяет неравенство состояний генераторов `d1` и `d2`
ostrm `<< d`	Записывает состояние генератора `d` в поток вывода *ostrm*

Окончание табл. 17.3

Операция	Описание
istrm >> d	Считывает новое состояние из потока ввода *istrm* в генератор d
distr::param_type	Тип параметров *distr*
distr d(*pt*)	Создает распределение с параметром param_type *pt*
d.param(*pt*)	Задает текущий параметр равным param_type *pt*
d.param()	Возвращает текущий параметр как значение типа param_type
d(*e*,*pt*)	Возвращает следующее значение с помощью генератора *e* и параметра param_type *pt* и переводит генератор в состояние *e*
d.*param*()	Возвращает значение параметра *param*

Распределение можно параметризовать, поэтому параметры *args* и *pt* зависят от конкретного распределения. Для параметризации можно

- передать список аргументов *args* конструктору;
- использовать член, соответствующий параметру, чтобы запросить его значение;
- использовать тип param_type, чтобы
  - передать эти параметры конструктору в виде одного аргумента;
  - передать эти параметры для генерации следующего значения;
  - запросить значения этих параметров.

Класс param_type должен иметь конструктор, которому можно передавать значения параметров, или именованные члены, возвращающие эти значения.

Например, равномерные распределения имеют два параметра, a и b, соответствующие минимуму и максимуму (см. раздел 17.1.5). В результате эти аргументы можно передавать по отдельности.

```
uniform_int_distribution<> d(0, 20); // инициализируем параметры "a" и "b"
d.a()                                // выясняем значение параметра "a"
d.b()                                // выясняем значение параметра "b"
d.param().a()                        // выясняем значение параметра "a"
d.param().b()                        // выясняем значение параметра "b"
```

Кроме того, можно просто передать объект класса param_type:

```
uniform_int_distribution<>::param_type pt(100, 200); // другая параметризация
d(e,pt)       // генерирует одно значение в соответствии с параметризацией pt
d.param(pt);  // позволяет всем сгенерированным значениям
              // использовать параметризацию  pt
```

Выборочные распределения используют специальные конструкторы для передачи векторов значений или генератора значений.

Отметим, что максимальное значение, передаваемое распределениям как параметр, иногда может включаться в интервал, а иногда нет.

Для генераторов случайных чисел состояние, записанное с помощью оператора <<, представляет собой десятичные значения, разделенные пробелами. Несмотря на то что они могут выглядеть как список случайных чисел, *это не так*.

Следующая программа демонстрирует использование распределений:

```
// num/dist1.cpp

#include <random>
#include <map>
#include <string>
#include <iostream>
#include <iomanip>

template <typename Distr, typename Eng>
void distr (Distr d, Eng e, const std::string& name)
{
   // выводим min, max и четыре значения
   std::cout << name << ":" << std::endl;
   std::cout << "- min():  " << d.min() << std::endl;
   std::cout << "- max():  " << d.max() << std::endl;
   std::cout << "- values: " << d(e) << ' ' << d(e) << ' '
             << d(e) << ' ' << d(e) << std::endl;

   // выводим сгенерированные значения (преобразованные в целые числа)
   std::map<long long,int> valuecounter;
   for (int i=0; i<200000; ++i) {
        valuecounter[d(e)]++;
   }

   // и распечатываем полученное распределение
   std::cout << "====" << std::endl;
   for (auto elem : valuecounter) {
        std::cout << std::setw(3) << elem.first << ": "
                  << elem.second << std::endl;
   }
   std::cout << "====" << std::endl;
   std::cout << std::endl;
}

int main()
{
   std::knuth_b e;

   std::uniform_real_distribution<> ud(0, 10);
   distr(ud,e,"uniform_real_distribution");

   std::normal_distribution<> nd;
   distr(nd,e,"normal_distribution");

   std::exponential_distribution<> ed;
   distr(ed,e,"exponential_distribution");

   std::gamma_distribution<> gd;
   distr(gd,e,"gamma_distribution");
}
```

На моем компьютере программа выдала следующий результат (представленный в двух столбцах):

```
uniform_real_distribution:
- min():   0
- max():   10
- values:  8.30965 1.30427 9.47764 3.83416
====
  0: 20087
  1: 20057
  2: 19878
  3: 19877
  4: 20005
  5: 20118
  6: 20063
  7: 19886
  8: 20003
  9: 20026
====
exponential_distribution:
- min():  0
- max():  1.79769e+308
- values: 0.185167 2.03694 0.0536495 0.958636
====
  0: 126487
  1: 46436
  2: 17120
  3: 6294
  4: 2326
  5: 865
  6: 283
  7: 107
  8: 52
  9: 17
  10: 6
  11: 6
  12: 1
====
```

```
normal_distribution:
- min():  2.22507e-308
- max():  1.79769e+308
- values: -0.131724 0.117963 -0.140331 0.538967
====
  -4: 9
  -3: 245
  -2: 4325
  -1: 26843
  0: 136947
  1: 26987
  2: 4377
  3: 258
  4: 9
====

gamma_distribution:
- min():  0
- max():  1.79769e+308
- values: 0.117964 1.60557 0.558526 1.21066
====
  0: 126315
  1: 46477
  2: 17160
  3: 6271
  4: 2413
  5: 866
  6: 327
  7: 109
  8: 41
  9: 12
  10: 7
  11: 1
  12: 1
====
```

17.1.5. Подробное описание распределений

Рассмотрим функции распределения, плотность вероятностей и их параметры.

uniform_int_distribution:

$$P(i|a,b)=\frac{1}{b-a+1}.$$

Параметры:

IntType a (по умолчанию: 0)
IntType b (по умолчанию: *limits*::max())

uniform_real_distribution:

$$P(x|a,b)=\frac{1}{b-a}$$

Параметры:

RealType **a** (по умолчанию: 0.0)
RealType **b** (по умолчанию: 1.0)

bernoulli_distribution:

$$P(b \mid p) = \begin{cases} p, & \text{если } b = \texttt{true}, \\ 1-p, & \text{если } b = \texttt{false}. \end{cases}$$

Параметры:
`double` **p** (по умолчанию: `0.5`)

binomial_distribution:

$$P(i \mid t, p) = \binom{t}{i} p^i (1-p)^{t-i}.$$

Параметры:
IntType **t** (по умолчанию: `1`)
`double` **p** (по умолчанию: `0.5`)

geometric_distribution:

$$P(i \mid p) = p(1-p)^i.$$

Параметры:
`double` **p** (по умолчанию: `0.5`)

negative_binomial_distribution:

$$P(i \mid k, p) = \binom{k+i-1}{i} p^k (1-p)^i.$$

Параметры:
IntType **k** (по умолчанию: `1`)
`double` **p** (по умолчанию: `0.5`)

poisson_distribution:

$$P(i \mid \mu) = \frac{e^{-\mu}\mu^i}{i!}.$$

Параметры:
`double` **mean** (по умолчанию: `1.0`)

exponential_distribution:

$$P(x \mid \lambda) = \lambda e^{-\lambda x}.$$

Параметры:
RealType **lambda** (по умолчанию: `1.0`)

gamma_distribution:

$$P(x|\alpha,\beta) = \frac{e^{-x/\beta}}{\beta^{\alpha}\Gamma(\alpha)} x^{\alpha-1}.$$

Параметры:
RealType **alpha** (по умолчанию: `1.0`)
RealType **beta** (по умолчанию: `1.0`)

weibull_distribution:

$$P(x|a,b)=\frac{a}{b}\left(\frac{x}{b}\right)^{a-1}\exp\left(-\left(\frac{x}{b}\right)^{a}\right).$$

Параметры:
RealType **a** (по умолчанию: `1.0`)
RealType **b** (по умолчанию: `1.0`)

extreme_value_distribution:

$$P(x|a,b)=\frac{1}{b}\exp\left(\frac{a-x}{b}-\exp\left(\frac{a-x}{b}\right)\right).$$

Параметры:
RealType **a** (по умолчанию: `1.0`)
RealType **b** (по умолчанию: `1.0`)

normal_distribution:

$$P(x|\mu,\sigma)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\left(\frac{x-\mu}{2\sigma^2}\right)^2\right).$$

Параметры:
RealType **mean** (по умолчанию: `0.0`)
RealType **stddev** (по умолчанию: `1.0`)

lognormal_distribution:

$$P(x|m,s)=\frac{1}{sx\sqrt{2\pi}}\exp\left(-\frac{(\ln x-m)^2}{2s^2}\right).$$

Параметры:
RealType **m** (по умолчанию: `0.0`)
RealType **s** (по умолчанию: `1.0`)

chi_squared_distribution:

$$P(x\,|\,n)=\frac{x^{(n/2)-1}e^{-x/2}}{\Gamma(n/2)2^{n/2}}.$$

Параметры:
RealType **n** (по умолчанию: `1`)

cauchy_distribution:

$$P(x|a,b)=\left(\pi b\left(1+\left(\frac{x-a}{b}\right)^2\right)\right)^{-1}.$$

Параметры:
RealType **a** (по умолчанию: `1.0`)
RealType **b** (по умолчанию: `1.0`)

fisher_f_distribution:

$$P(x|m,n)=\frac{\Gamma\left((m+n)/2\right)}{\Gamma(m/2)\Gamma(n/2)}\left(\frac{m}{n}\right)^{m/2}x^{(m/2)-1}\left(1+\frac{mx}{n}\right)^{-(m+n)/2}.$$

Параметры:
RealType **m** (по умолчанию: 1)
RealType **n** (по умолчанию: 1)

student_t_distribution:

$$p(x|n)=\frac{1}{\sqrt{n\pi}}\frac{\Gamma\left((n+1)/2\right)}{\Gamma(n/2)}\left(1+\frac{x^2}{n}\right)^{-(n+1)/2}.$$

Параметры:
RealType **n** (по умолчанию: 1)

discrete_distribution:

$$P(i|p_0,\ldots,p_{n-1})=p_i.$$

Параметры:
`vector<double>` **probabilities**

Конструкторы получают списки инициализации, состоящие из чисел типа `double`, интервал чисел типа `double` и функциональный объект, создающий числа типа `double`.

piecewise_constant_distribution:

$$p(x|b_0,\ldots,b_n,\rho_0,\ldots,\rho_{n-1})=\rho_i \text{ для } b_i\le x<b_{i+1}.$$

Параметры:
`vector<`*RealType*`>` **intervals** (по умолчанию: не предусмотрено)
`vector<`*RealType*`>` **densities** (по умолчанию: не предусмотрено)

Конструктор получает интервал граничных значений и весов, список инициализации границ и генератор весов, а также число, минимум и максимум границ и генератор весов.

piecewise_linear_distribution:

$$p(x|b_0,\ldots,b_n,\rho_0,\ldots,\rho_{n-1})=\rho_i\frac{b_{i+1}-x}{b_{i+1}-b_i}+\rho_{i+1}\frac{x-b_i}{b_{i+1}-b_i} \text{ для } b_i\le x<b_{i+1}.$$

Параметры:
`vector<`*RealType*`>` **intervals**
`vector<`*RealType*`>` **densities**

Конструктор получает интервал граничных значений и весов, список инициализации границ и генератор весов, а также число, минимум и максимум границ и генератор весов.

Дальнейшие детали можно найти в спецификации стандартной библиотеки C++.

17.2. Комплексные числа

Стандартная библиотека C++ содержит шаблонный класс `complex<>`, предназначенный для работы с комплексными числами. Комплексные числа состоят из двух частей: действительной и мнимой, причем квадрат мнимой части является отрицательным числом. Иначе говоря, мнимая часть комплексного числа — это множитель перед квадратным корнем из −1, который обозначается символом *i*.

Изменения, внесенные стандартом C++11

Стандарт C++98 определял практически все свойства класса `complex<>`. Они были определены уже при первой стандартизации языка C++. За исключением ключевого слова `constexpr` ни одно из новых функциональных свойств языка C++11 не оказало влияния на класс `complex<>` и его специализации. Было исправлено лишь несколько неточностей и добавлено небольшое количество усовершенствований.

- Добавлены операции, получающие комплексные числа как аргументы: `proj()`, `asin()`, `acos()`, `atan()`, `asin()`, `acosh()` и `atanh()`.
- Теперь с помощью функций `real(`*val*`)` и `imag(`*val*`)` действительную и мнимую части комплексного числа можно задать непосредственно.

17.2.1. Общее описание класса complex<>

Класс `complex<>` объявлен в заголовочном файле `<complex>`:

```
#include <complex>
```

В заголовочном файле `<complex>` класс `complex<>` определен следующим образом:

```
namespace std {
   template <typename T>
   class complex;
}
```

Шаблонный параметр `T` используется как скалярный тип и действительной, и мнимой части комплексного числа.

Кроме того, стандартная библиотека C++ содержит три специализации для типов `float`, `double` и `long double`:

```
namespace std {
   template<> class complex<float>;
   template<> class complex<double>;
   template<> class complex<long double>;
}
```

Эти типы позволяют выполнить определенную оптимизацию и безопасное преобразование одного типа комплексного числа в другой.

17.2.2. Примеры использования класса complex<>

Следующая программа демонстрирует некоторые из возможностей класса complex<> по созданию комплексных чисел, их выводу в разных представлениях и выполнению некоторых типичных операций над комплексными числами:

```
// num/complex1.cpp

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

int main()
{
   // комплексное числа с действительной и мнимой частями
   // - действительная часть: 4.0
   // - мнимая часть: 3.0
   complex<double> c1(4.0,3.0);

   // создаем комплексное число в полярных координатах
   // - модуль: 5.0
   // - аргумент: 0.75
   complex<float> c2(polar(5.0,0.75));

   // выводим комплексные числа с действительной и мнимой частями
   cout << "c1: " << c1 << endl;
   cout << "c2: " << c2 << endl;

   // выводим комплексные числа в полярных координатах
   cout << "c1: magnitude: " << abs(c1)
        << " (squared magnitude: " << norm(c1) << ") "
        << " phase angle: " << arg(c1) << endl;
   cout << "c2: magnitude: " << abs(c2)
        << " (squared magnitude: " << norm(c2) << ") "
        << " phase angle: " << arg(c2) << endl;

   // выводим сопряженные комплексные числа
   cout << "c1 conjugated: " << conj(c1) << endl;
   cout << "c2 conjugated: " << conj(c2) << endl;

   // выводим результат вычислений
   cout << "4.4 + c1 * 1.8: " << 4.4 + c1 * 1.8 << endl;

   // выводим сумму c1 и c2:
   // - примечание: типы разные
   cout << "c1 + c2: "
        << c1 + complex<double>(c2.real(),c2.imag()) << endl;

   // добавляем к c1 квадратный корень из c1 и выводим результат
   cout << "c1 += sqrt(c1): " << (c1 += sqrt(c1)) << endl;
}
```

Эта программа может вывести следующий результат (его точный вид зависит от конкретной реализации типа double):

```
c1: (4,3)
c2: (3.65844,3.40819)
c1: magnitude: 5 (squared magnitude: 25) phase angle: 0.643501
c2: magnitude: 5 (squared magnitude: 25) phase angle: 0.75
c1 conjugated: (4,-3)
c2 conjugated: (3.65844,-3.40819)
4.4 + c1 * 1.8: (11.6,5.4)
c1 + c2: (7.65844,6.40819)
c1 += sqrt(c1): (6.12132,3.70711)
```

Второй пример содержит цикл, в котором вводятся два комплексных числа и первое число возводится в степень второго:

```
// num/complex2.cpp

#include <iostream>
#include <complex>
#include <cstdlib>
#include <limits>


#include <cstdio>

using namespace std;
int main()
{
   complex<long double> c1, c2;
   while (cin.peek() != EOF) {

      // вводим первое комплексное число
      cout << "complex number c1: ";
      cin >> c1;
      if (!cin) {
          cerr << "input error" << endl;
          return EXIT_FAILURE;
      }

      // вводим второе комплексное число
      cout << "complex number c2: ";
      cin >> c2;

      if (!cin) {
          cerr << "input error" << endl;
          return EXIT_FAILURE;
      }

      if (c1 == c2) {
          cout << "c1 and c2 are equal !" << endl;
      }

      cout << "c1 raised to the c2: " << pow(c1,c2)
```

```
            << endl << endl;

        // пропускаем остальную часть строки
        cin.ignore(numeric_limits<int>::max(),'\n');
    }
}
```

В табл. 17.4 приведены возможные ввод и вывод программы. Отметим, что комплексное число можно задавать в виде отдельной действительной части в скобках или без них, а также в виде пары чисел, представляющих собой действительную и мнимую части, заключенные в скобки и разделенные запятыми.

Таблица 17.4. Возможный ввод и вывод программы complex2.cpp

c1	c2	Вывод
2	2	c1 raised to c2: (4,0)
(16)	0.5	c1 raised to c2: (4,0)
(8,0)	0.333333333	c1 raised to c2: (2,0)
0.99	(5)	c1 raised to c2: (0.95099,0)
(0,2)	2	c1 raised to c2: (-4,4.89843e-16)
(1.7,0.3)	0	c1 raised to c2: (1,0)
(3,4)	(-4,3)	c1 raised to c2: (4.32424e-05,8.91396e-05)
(1.7,0.3)	(4.3,2.8)	c1 raised to c2: (-4.17622,4.86871)

17.2.3. Операции над комплексными числами

Шаблонный класс complex<> содержит операции, описанные в следующих подразделах.

Создание, копирование и присваивание

В табл. 17.5 приведены конструкторы и операции присваивания для класса complex. Конструкторы могут получать начальные значения действительной и мнимой частей. Если эти части не заданы, они инициализируются конструктором по умолчанию в зависимости от типа их значений.

Таблица 17.5. Конструкторы и операции присваивания для класса complex<>

Выражение	Описание
complex c	Создает комплексное число с нулевыми действительной и мнимой частями, т.е. 0 + 0*i*
complex c(1.3)	Создает комплексное число, у которого действительная часть равна 1.3, а мнимая — 0, т.е. 1.3 + 0*i*
complex c(1.3,4.2)	Создает комплексное число, у которого действительная часть равна 1.3, а мнимая — 4.2, т.е. 1.3 + 4.2*i*
complex c1(c2)	Создает комплексное число c1 как копию c2

Окончание табл. 17.5

Выражение	Описание
`polar(4.2)`	Создает временное комплексное число в полярных координатах (модуль *rho* равен `4.2`, а аргумент *theta* — `0`)
`polar(4.2,0.75)`	Создает временное комплексное число в полярных координатах (модуль *rho* равен `4.2`, а аргумент *theta* — `0,75`)
`conj(c)`	Создает временное комплексное число, которое является сопряженным к комплексному числу `c` (комплексное число с противоположной по знаку мнимой часть)
`proj(c)`	Создает временное комплексное число по проекции числа `c` на сферу Римана (начиная со стандарта C++11)
`c1 = c2`	Присваивает число `c2` числу `c1`
`c1 += c2`	Добавляет число `c2` к числу `c1`
`c1 -= c2`	Вычитает число `c2` из числа `c1`
`c1 *= c2`	Умножает число `c2` на число `c1` и присваивает результат числу `c1`
`c1 /= c2`	Делит число `c1` на число `c2` и присваивает результат числу `c1`

Значение существующего комплексного числа можно изменить только с помощью операторов присваивания. Сокращенные операторы присваивания `+=`, `-=`, `*=` и `/=` суммируют, вычитают, умножают и делят два комплексных числа, записывая результат в первый операнд.

Вспомогательная функция `polar()` позволяет создать комплексное число, инициализированное полярными координатами (модуль и аргумент, заданный в радианах):

```
// создаем комплексное число, инициализированное полярными координатами
std::complex<double> c2(std::polar(4.2,0.75));
```

При неявном преобразовании комплексных чисел во время их создания возникает одна проблема. Например, следующее выражение является корректным:

```
std::complex<float> c2(std::polar(4.2,0.75));   // OK
```

Однако конструкция, приведенная ниже, является ошибочной.

```
std::complex<float> c2 = std::polar(4.2,0.75);  // ОШИБКА
```

Эта проблема рассматривается в следующем подразделе.

Вспомогательная функция `conj()` позволяет создать комплексное число, инициализированное значением, сопряженным с другим комплексным числом (т. е. комплексным числом с противоположным знаком мнимой части).

```
std::complex<double> c1(1.1,5.5);
std::complex<double> c2(conj(c1)); // инициализируем число c2 комплексным числом
                                   // complex<double>(1.1,-5.5)
```

В стандарте C++11 введена вспомогательная функция `proj()`, создающая комплексное число, представляющее собой проекцию другого комплексного числа на сферу Римана.

Неявные преобразования типов

Конструкторы для специализаций `float`, `double` и `long double` разработаны так, чтобы безопасные преобразования, например `complex<float>` в `complex<double>`, можно было выполнять неявно, а менее безопасные, такие как `complex<long double>` в `complex<double>`, только явно.

```
std::complex<float> cf;
std::complex<double> cd;
std::complex<long double> cld;
...
std::complex<double> cd1 = cf;  // OK: безопасное преобразование
std::complex<double> cd2 = cld; // ОШИБКА: не неявное преобразование
std::complex<double> cd3(cld);  // OK: явное преобразование
```

Кроме того, не существует конструкторов, создающих комплексное число из комплексного числа другого типа. В частности, нельзя преобразовать тип `complex` с целочисленными действительной и мнимой частями в тип `complex` с действительной и мнимой частями типа `float`, `double` или `long double`. Однако эти преобразования можно выполнять при передаче вещественной и мнимой частей в виде отдельных аргументов.

```
std::complex<double> cd;
std::complex<int> ci;
...
std::complex<double> cd4 = ci; // ОШИБКА: неявное преобразование не допускается
std::complex<double> cd5(ci);  // ОШИБКА: явное преобразование не допускается
std::complex<double> cd6(ci.real(),ci.imag()); // OK
```

К сожалению, операции присваивания позволяют выполнять менее безопасные преобразования и определены в виде шаблонных функций для всех типов. Следовательно, можно присваивать комплексные числа любых типов, если типы значений действительной и мнимой частей допускают преобразования[7].

```
std::complex<double> cd;
std::complex<long double> cld;


std::complex<int> ci;
...
cd = ci;  // OK
cd = cld; // OK
```

Эта проблема относится также к функциям `polar()`, `conj()` и `proj()`. Например, следующая конструкция работает отлично:

```
std::complex<float> c2(std::polar(4.2,0.75));  // OK
```

Однако конструкция с операцией присваивания является ошибочной:

```
std::complex<float> c2 = std::polar(4.2,0.75); // ОШИБКА
```

[7] То, что конструкторы специализаций комплексных чисел допускают только безопасные неявные преобразования, а операции присваивания допускают любые неявные преобразования, вероятно, является ошибкой разработчиков стандарта.

Причина заключается в том, что выражение

```
std::polar(4.2,0.75)
```

создает временный объект класса `complex<double>`, а неявное преобразование типа `complex<double>` в тип `complex<float>` не определено[8].

Доступ к значениям

В табл. 17.6 перечислены функции, обеспечивающие доступ к атрибутам комплексных чисел.

Таблица 17.6. Операции доступа к членам класса `complex<>`

Выражение	Описание
`real(c)`	Возвращает значение действительной части (как глобальная функция)
`c.real()`	Возвращает значение действительной части (как функция-член класса)
`c.real(1.7)`	Присваивает `1.7` как новую действительную часть (начиная со стандарта C++11)
`imag(c)`	Возвращает значение мнимой части (как глобальная функция)
`c.imag()`	Возвращает значение мнимой части (как функция-член класса)
`c.imag(1.7)`	Присваивает `1.7` как новую мнимую часть (начиная со стандарта C++11)
`abs(c)`	Возвращает абсолютное значение числа `c` ($\sqrt{\text{c.real()}^2 + \text{c.imag()}^2}$)
`norm(c)`	Возвращает абсолютное значения числа `c`, возведенное в квадрат ($\text{c.real()}^2 + \text{c.imag()}^2$)
`arg(c)`	Возвращает аргумент j представления комплексного числа `c` в полярных координатах (эквивалент `atan2(c.imag(),c.real())`)

До принятия стандарта C++11 функции `real()` и `imag()` обеспечивали только доступ к действительной и мнимой частям комплексного числа. Изменить только действительную или только мнимую части комплексного числа можно было, только присвоив ему новое комплексное число. Например, следующий оператор задает мнимую часть числа `c` равной `3.7`:

```
std::complex<double> c;
...
c = std::complex<double>(c.real(),3.7); // начиная со стандарта C++11: c.imag(3.7)
```

Операции сравнения

Для сравнения комплексных чисел предусмотрены только проверки равенства и неравенства (табл. 17.7). Операторы == и != определены как глобальные функции, поэтому один из операндов может быть скалярным значением. Операнд, заданный скалярным

[8] В общем случае синтаксис инициализации с оператором = требует, чтобы было возможным неявное преобразование типов.

значением, интерпретируется как действительная часть, а мнимая часть по умолчанию равна нулю, определенным для данного типа (обычно 0).

Таблица 17.7. Операции сравнения в классе complex<>

Выражение	Описание
`c1 == c2`	Проверяет равенство c1 и c2 (`c1.real()==c2.real() && c1.imag()==c2.imag()`)
`c == 1.7`	Проверяет равенство c и 1.7 (`c.real()==1.7 && c.imag()==0.0`)
`1.7 == c`	Проверяет равенство c и 1.7 (`c.real()==1.7 && c.imag()==0.0`)
`c1 != c2`	Проверяет неравенство c1 и c2 (`c1.real()!=c2.real() || c1.imag()!=c2.imag()`)
`c != 1.7`	Проверяет неравенство c и 1.7 (`c.real()!=1.7 || c.imag()!=0.0`)
`1.7 != c`	Проверяет неравенство c и 1.7 (`c.real()!=1.7 || c.imag()!=0.0`)

Другие операторы сравнения, например <, не определены. Несмотря на то что для комплексных чисел теоретически можно определить отношение "больше", такое упорядочение оказывается искусственным и практически бесполезным. Например, бессмысленно сравнивать комплексные числа по модулю, потому что два совершенно разных комплексных числа (например, 1 и -1) могут иметь одинаковые модули. Впрочем, можно придумать вполне корректное отношение "больше". Например, можно считать, что комплексное число c1 меньше комплексного числа c2, если `|c1|<|c2|`, а при равенстве модулей, если `arg(c1)<arg(c2)`. Однако такой критерий практически не имеет математического смысла[9].

В результате тип `complex<>` невозможно использовать в качестве типа элемента ассоциативного контейнера, если не задать пользовательский критерий сортировки. Причина заключается в том, что ассоциативные контейнеры используют функциональный объект `less<>`, который вызывает оператор < для сортировки элементов (см. раздел 6.11.1).

Реализовав пользовательский оператор <, можно сортировать комплексные числа и хранить их в ассоциативных контейнерах. При этом следует быть очень осторожным и не засорять стандартное пространство имен.

Рассмотрим пример:

```
template <typename T>
bool operator< (const std::complex<T>& c1,
                const std::complex<T>& c2)
{
   return std::abs(c1)<std::abs(c2) ||
          (std::abs(c1)==std::abs(c2) &&
          std::arg(c1)<std::arg(c2));
}
```

Арифметические операции

Для комплексных чисел определены четыре основные арифметические операции, а также операции применения знаков "плюс" и "минус" (табл. 17.8).

[9] Благодарю за это замечание Дэвида Вандевурда (David Vandevoorde).

Таблица 17.8. Арифметические операции над классом complex<>

Выражение	Описание
`c1 + c2`	Возвращает сумму `c1` и `c2`
`c + 1.7`	Возвращает сумму `c` и `1.7`
`1.7 + c`	Возвращает сумму `1.7` и `c`
`c1 - c2`	Возвращает разность `c1` и `c2`
`c - 1.7`	Возвращает разность `c` и `1.7`
`1.7 - c`	Возвращает разность `1.7` и `c`
`c1 * c2`	Возвращает произведение `c1` и `c2`
`c * 1.7`	Возвращает произведение `c` и `1.7`
`1.7 * c`	Возвращает произведение `1.7` и `c`
`c1 / c2`	Возвращает частное от деления `c1` на `c2`
`c / 1.7`	Возвращает частное от деления `c` на `1.7`
`1.7 / c`	Возвращает частное от деления `1.7` на `c`
`- c`	Возвращает число `c` с противоположным знаком
`+ c`	Возвращает число `c`
`c1 += c2`	Эквивалент `c1 = c1 + c2`
`c1 -= c2`	Эквивалент `c1 = c1 - c2`
`c1 *= c2`	Эквивалент `c1 = c1 * c2`
`c1 /= c2`	Эквивалент `c1 = c1 / c2`

Операции ввода-вывода

Класс `complex` поддерживает общие операторы ввода-вывода << и >> (табл. 17.9).

Оператор вывода записывает комплексное число в соответствии с текущим состоянием потока, используя формат:

(*realpart*, *imagpart*)

Таблица 17.9. Операции ввода-вывода для класса complex<>

Выражение	Описание
`strm << c`	Записывает комплексное число `c` в поток вывода `strm`
`strm >> c`	Считывает комплексное число `c` из потока ввода `strm`

В частности, операция вывода является эквивалентом следующей реализации:

```
template <typename T, typename charT, typename traits>
std::basic_ostream<charT,traits>&
operator << (std::basic_ostream<charT,traits>& strm,
             const std::complex<T>& c)
{
   // временная строка значений для вывода с одним аргументом
```

```
    std::basic_ostringstream<charT,traits> s;
    s.flags(strm.flags());          // копируем флаги потока
    s.imbue(strm.getloc());         // копируем локальный контекст потока
    s.precision(strm.precision()); // копируем точность потока

    // подготавливаем строку значений
    s << '(' << c.real() << ',' << c.imag() << ')';

    // записываем строку значений
    strm << s.str();
    return strm;
}
```

Операция ввода дает возможность читать комплексное число в одном из следующих форматов:

(realpart, imagpart)
(realpart)
realpart

Если ни один из следующих символов не соответствует ни одному из форматов, то устанавливается флаг `ios::failbit`, что может привести к генерированию соответствующего исключения (см. раздел 15.4.4).

К сожалению, задать разделитель между действительной и мнимой частями комплексного числа невозможно. По этой причине, если вы используете в качестве "десятичной точки" запятую (как в Германии), то ввод-вывод будет выглядеть очень странно. Например, комплексное число с действительной частью `4.6` и мнимой частью `2.7` записывается в виде

```
(4,6,2,7).
```

Пример использования операций ввода-вывода приведен в разделе 17.2.2.

Трансцендентные функции

В табл. 17.10 перечислены трансцендентные функции (тригонометрические, экспоненциальные и др.) для класса `complex`.

Таблица 17.10. Трансцендентные функции для класса `complex<>`

Выражение	Описание
`pow(c,3)`	Возведение комплексного числа в степень: c^3
`pow(c,1.7)`	Возведение комплексного числа в степень: $c^{1.7}$
`pow(c1,c2)`	Возведение комплексного числа в степень: $c1^{c2}$
`pow(1.7,c)`	Возведение комплексного числа в степень: 1.7^c
`exp(c)`	Возведение в степень `c` по основанию *e*: e^c
`sqrt(c)`	Квадратный корень из `c`: ()
`log(c)`	Комплексный натуральный логарифм `c` по основанию *e* (`ln c`)
`log10(c)`	Комплексный десятичный логарифм `c` (`lg c`)
`sin(c)`	Синус `c` (`sin c`)

Окончание табл. 7.10

Выражение	Описание
`cos(c)`	Косинус `c` (`cos c`)
`tan(c)`	Тангенс `c` (`tan c`)
`sinh(c)`	Гиперболический синус `c` (`sinh c`)
`cosh(c)`	Гиперболический косинус `c` (`cosh c`)
`tanh(c)`	Гиперболический тангенс `c` (`tanh c`)
`asin(c)`	Арксинус `c` (начиная со стандарта C++11)
`acos(c)`	Арккосинус `c` (начиная со стандарта C++11)
`atan(c)`	Арктангенс `c` (начиная со стандарта C++11)
`asinh(c)`	Гиперболический арксинус `c` (начиная со стандарта C++11)
`acosh(c)`	Гиперболический арккосинус `c` (начиная со стандарта C++11)
`atanh(c)`	Гиперболический арктангенс `c` (начиная со стандарта C++11)

17.2.4. Подробное описание класса `complex<>`

В этом подразделе подробно описываются все операции класса `complex<>`. В следующих определениях символ `T` обозначает шаблонный параметр класса `complex<>`, являющийся типом действительной и мнимой частей значения класса `complex`.

Определения типов

`complex::value_type`

- Тип действительной и мнимой частей.

Создание, копирование и присваивание

`complex::complex ()`

- Конструктор по умолчанию.
- Создает комплексное число, в котором действительная и мнимая части инициализируются с помощью явного вызова их конструктора по умолчанию. Таким образом, для элементарных типов начальное значение действительной и мнимой частей равно `0` (подробно о значениях, заданных по умолчанию для элементарных типов, см. в разделе 3.2.1).

`complex::complex (const T& re)`

- Создает комплексное число, в котором аргумент *re* является значением действительной части, а мнимая часть инициализируется с помощью явного вызова ее конструктора по умолчанию (`0` для элементарных типов).
- Этот конструктор также определяет автоматическое преобразования `T` в `complex`.

complex::complex (const T& *re*, const T& *im*)

- Создает комплексное число, в котором аргумент *re* является действительной частью, а *im* — мнимой.

complex polar (const T& *rho*)
complex polar (const T& *rho*, const T& *theta*)

- Обе версии создают и возвращают комплексное число, инициализированное полярными координатами.
- Аргумент *rho* задает модуль.
- Аргумент *theta* задает аргумент в радианах (по умолчанию: 0).

complex conj (const *complex*& *cmplx*)

- Создает и возвращает комплексное число, инициализированное комплексным числом, сопряженным с *cmplx* (мнимая часть имеет противоположный знак).

complex proj (const *complex*& *cmplx*)

- Создает и возвращает временное комплексное число на основе проекции x на сферу Римана.
- Эквивалент функции cproj() из языка C.
- Доступна по стандарту C++11.

complex::complex (const *complex*& *cmplx*)

- Копирующий конструктор.
- Создает новое комплексное число в виде копии аргумента *cmplx*.
- Копирует действительную и мнимую части.
- Эта функция существует как в шаблонном, так и в не шаблонном варианте (см. раздел 3.2). Таким образом, она поддерживает автоматическое преобразование элементарных типов.
- Однако конструктор по умолчанию ограничен специализациями для типов float, double и long double, поэтому менее безопасные преобразования — из double и long double в float, а также из long double в double — в неявном виде невозможны. Более подробно этот вопрос рассмотрен в разделе 17.2.3.

complex& complex::operator= (const *complex*& *cmplx*)

- Присваивает комплексное число *cmplx*.
- Возвращает *this.
- Эта функция существует как в шаблонном, так и в не шаблонном варианте (см. раздел 3.2). Таким образом, она поддерживает автоматическое преобразование элементарных типов. (Это относится и к специализациям, предусмотренным стандартной библиотекой C++.)

complex& **complex**::**operator** += (const *complex*& *cmplx*)
complex& **complex**::**operator** -= (const *complex*& *cmplx*)
complex& **complex**::**operator** *= (const *complex*& *cmplx*)
complex& **complex**::**operator** /= (const *complex*& *cmplx*)

- Эти операции вычисляют сумму, разность, произведение и частное от деления комплексных чисел *cmplx* и `*this` соответственно, записывая результат в `*this`.
- Возвращает `*this`.
- Эти функции существует как шаблонном, так и в не шаблонном варианте (см. раздел 3.2). Таким образом, она поддерживает автоматическое преобразование элементарных типов. (Это относится и к специализациям, предусмотренным стандартной библиотекой C++.)

Отметим, что операции присваивания — это единственные функции, позволяющие модифицировать значение существующего объекта класса `complex`.

Доступ к элементам

T **complex**::**real** () const
T **real** (const *complex*& *cmplx*)
T **complex**::**imag** () const
T **imag** (const *complex*& *cmplx*)

- Эти функции возвращают действительную или мнимую часть соответственно.
- Отметим, что возвращаемое значение не является ссылкой. Таким образом, эти функции нельзя использовать для модификации действительной и мнимой частей. Для изменения только действительной или только мнимой части необходимо присвоить новое комплексное число (см. раздел 17.2.3).

T **complex**::**real** (const T& *re*)
T **complex**::imag (const T& *im*)

- Эти функции присваивают аргументы *re* или *im* как новую действительную или мнимую часть соответственно.
- Доступны начиная со стандарта C++11. До принятия стандарта C++11 для модификации только действительной или только мнимой части приходилось присваивать новое комплексное число (см. раздел 17.2.3).

T **abs** (const *complex*& *cmplx*)

- Возвращает абсолютное значение (модуль) аргумента *cmplx*.
- Абсолютное значение равно $\sqrt{cmplx.\text{real}()^2 + cmplx.\text{imag}()^2}$.

T **norm** (const *complex*& *cmplx*)

- Возвращает модуль числа *cmplx* в квадрате.
- Квадрат абсолютного значения равен $cmplx.\text{real}()^2 + cmplx.\text{imag}()^2$.

T **arg** (const *complex*& *cmplx*)

- Возвращает аргумент (j) числа *cmplx* в полярной системе координат.
- Эквивалент функции atan2(*cmplx*.imag(),*cmplx*.real()).

Операции ввода-вывода

ostream& **operator** << (*ostream*& *strm*, const *complex*& *cmplx*)

- Записывает значение *cmplx* в поток *strm*, используя формат (*realpart*, *imagpart*)
- Возвращает *strm*.
- Точное описание этой операции см. в разделе 17.2.3.

istream& **operator** >> (*istream*& *strm*, *complex*& *cmplx*)

- Считывает новое значение из потока *strm* в *cmplx*.
- Корректными являются форматы

(*realpart*, *imagpart*)
(*realpart*)
realpart

- Возвращение *strm*.
- Точное описание этой операции см. в разделе 17.2.3.

Операции

complex **operator** + (const *complex*& *cmplx*)

- Положительный знак.
- Возвращает *cmplx*.

complex operator - (const *complex*& *cmplx*)

- Отрицательный знак.
- Возвращает значение *cmplx* с отрицательными действительной и мнимой частями.

complex binary-op (const *complex*& *cmplx1*, const *complex*& *cmplx2*)
complex binary-op (const *complex*& *cmplx*, const T& *value*)
complex binary-op (const T& *value*, const *complex*& *cmplx*)

- Все версии возвращают комплексное число, являющееся результатом операции **binary-op**.
- Операция **binary-op** может быть одной из следующий:

```
operator +
operator -
operator *
operator /
```

- Если в качестве типа элемента задается скалярное значение, оно интерпретируется как действительная часть, а мнимая часть имеет значение, заданное по умолчанию (0 для элементарных типов).

`bool` **comparison** (`const` *complex*`&` *cmplx1*, `const` *complex*`&` *cmplx2*)
`bool` **comparison** (`const` *complex*`&` *cmplx*, `const T&` *value*)
`bool` **comparison** (`const T&` *value*, `const` *complex*`&` *cmplx*)

- Возвращает результат сравнения двух комплексных чисел или результат сравнения комплексного числа со скалярным значением.
- Операция **comparison** может быть одной из двух:

```
operator ==
operator !=
```

- Если в качестве типа элемента задается скалярное значение, оно интерпретируется как действительная часть, а мнимая часть имеет значение, заданное по умолчанию (0 для элементарных типов).
- Операторы <, <=, > и >= не предусмотрены.

Трансцендентные функции

complex pow (`const` *complex*`&` *base*, `const T&` *exp*)
complex pow (`const` *complex*`&` *base*, `const` *complex*`&` *exp*)
complex pow (`const T&` *base*, `const` *complex*`&` *exp*)

- Все версии возвращают комплексное число *base* в степени *exp*:

  `exp(`*exp*`*log(`*base*`))`

- Точки ветвления лежат на отрицательном луче действительной прямой.
- Значение, возвращаемое функцией `pow(0,0)`, зависит от реализации.

complex exp (`const` *complex*`&` *cmplx*)

- Возвращает комплексное число *e* в степени *cmplx*.

complex sqrt (`const` *complex*`&` *cmplx*)

- Возвращает квадратный корень комплексного числа *cmplx*, лежащий в правой полуплоскости.
- Если аргумент является отрицательным действительным числом, то значение лежит на положительном луче мнимой оси.
- Точки ветвления лежат на отрицательном луче действительной прямой.

complex log (`const` *complex*`&` *cmplx*)

- Возвращает натуральный логарифм числа *cmplx*.
- Если *cmplx* является отрицательным действительным числом, то `imag(log(`*cmplx*`))` равно p.
- Точки ветвления лежат на отрицательном луче действительной прямой.

complex log10 (const *complex*& *cmplx*)

- Возвращает десятичный логарифм числа *cmplx*.
- Эквивалент `log(`*cmplx*`)/log(10)`.
- Точки ветвления лежат на отрицательном луче действительной прямой.

complex sin (const *complex*& *cmplx*)
complex cos (const *complex*& *cmplx*)
complex tan (const *complex*& *cmplx*)
complex sinh (const *complex*& *cmplx*)
complex cosh (const *complex*& *cmplx*)
complex tanh (const *complex*& *cmplx*)
complex asin (const *complex*& *cmplx*)
complex acos (const *complex*& *cmplx*)
complex atan (const *complex*& *cmplx*)
complex asinh (const *complex*& *cmplx*)
complex acosh (const *complex*& *cmplx*)
complex atanh (const *complex*& *cmplx*)

- Эти операции возвращают значение соответствующей тригонометрической функции от аргумента *cmplx*.
- Обратные тригонометрические операции (имя которых начинается с буквы **a**) доступны начиная со стандарта C++11.

17.3. Глобальные числовые функции

Заголовочные файлы <cmath> и <cstdlib> содержат определения функций, унаследованных от языка C (табл. 17.11 и 17.12)[10].

Таблица 17.11. Функции заголовочного файла <cmath>

Функция	Описание
pow()	Возведение в степень
exp()	Экспонента
sqrt()	Квадратный корень
log()	Натуральный логарифм
log10()	Десятичный логарифм
sin()	Синус
cos()	Косинус
tan()	Тангенс
sinh()	Гиперболический синус

[10] По историческим причинам некоторые числовые функции определены в заголовочном файле <cstdlib>, а не в <cmath>.

Окончание табл. 17.11

Функция	Описание
`cosh()`	Гиперболический косинус
`tanh()`	Гиперболический тангенс
`asin()`	Арксинус
`acos()`	Арккосинус
`atan()`	Арктангенс
`atan2()`	Арктангенс частного
`asinh()`	Гиперболический арксинус (начиная со стандарта C++11)
`acosh()`	Гиперболический косинус (начиная со стандарта C++11)
`atanh()`	Гиперболический арктангенс (начиная со стандарта C++11)
`ceil()`	Число с плавающей точкой, округленное для следующего целого числа
`floor()`	Число с плавающей точкой, округленное для предыдущего целого числа
`fabs()`	Абсолютное значение числа с плавающей точкой
`fmod()`	Остаток от деления чисел с плавающей точкой (по модулю)
`frexp()`	Преобразует число с плавающей точкой в целую и дробную части
`ldexp()`	Умножает число с плавающей точкой на степень двойки
`modf()`	Выделяет из числа с плавающей точкой целую и дробную части со знаками

В отличие от языка C, в языке C++ некоторые операции перегружены для разных типов. В итоге некоторые числовые функции из языка C оказались устаревшими. Например, в языке C есть функции `abs()`, `labs()`, `llabs()`, `fabs()`, `fabsf()` и `fabsl()` для обработки абсолютных значений типа `int`, `long`, `long long`, `double`, `float()` и `long double` соответственно. В языке C++ функция `abs()` является перегруженной, поэтому ее можно применять ко всем указанным выше типам.

Таблица 17.12. Числовые функции из заголовочного файла <cstdlib>

Функция	Описание
`abs()`	Абсолютное значение типа `int`
`labs()`	Абсолютное значение типа `long`
`llabs()`	Абсолютное значение типа `long long` (начиная со стандарта C++11)
`div()`	Частное и остаток от деления чисел типа `int`
`ldiv()`	Частное и остаток от деления чисел типа `long`
`lldiv()`	Частное и остаток от деления чисел типа `long long` (начиная со стандарта C++11)
`srand()`	Генератор случайных чисел (создает начальное значение для новой последовательности)
`rand()`	Генератор случайных чисел (создает следующее число в последовательности)

В частности, все числовые функции для целочисленных значений перегружены для типов `int`, `long` и `long long`, а все числовые функции для чисел с плавающей точкой перегружены для типов `float`, `double` и `long double`.

Однако у перегрузки есть важный побочный эффект: если передать целочисленное значение функции, для которой существует только версия, перегруженная для чисел с плавающей точкой, то выражение станет неоднозначным[11]:

```
std::sqrt(7) // НЕОДНОЗНАЧНОСТЬ: sqrt(float), sqrt(double) или
             // sqrt(long double)?
```

Вместо этого следует писать

```
std::sqrt(7.0) // OK
```

или, если используется переменная,

```
int x;
...
std::sqrt(float(x)) // OK
```

Разработчики библиотек решают эту проблему по-разному: одни используют перегрузку, другие поддерживают стандартное поведение (перегрузку для всех типов с плавающей точкой), третьи реализуют перегрузку для всех числовых типов вообще, а некоторые позволяют выбирать стратегию в зависимости от препроцессора. Таким образом, на практике неоднозначность может возникнуть, а может и не возникнуть. Для того чтобы написать переносимый код, всегда следует писать программу так, чтобы аргументы точно соответствовали ожидаемым типам.

17.4. Массивы значений

Еще со времен стандарта C++98 стандартная библиотека C++ содержит класс `valarray`, предназначенный для обработки массивов числовых значений.

Предназначение массивов значений

Массив значений реализует математическую концепцию линейной последовательности значений. Он имеет одно измерение, но с помощью специальных приемов вычисления индексов и мощных средств выделения подмножеств создает иллюзию большей размерности. Следовательно, массив значений можно использовать как для векторных, так и для матричных вычислений, а также для эффективной обработки систем полиномиальных уравнений.

С формальной точки зрения массивы значений — это одномерные массивы элементов, последовательно пронумерованных начиная с нуля. Массивы значений позволяют выполнять операции над всеми элементами или подмножествами одного или нескольких массивов значений. Например, можно вычислить выражение

```
z = a*x*x + b*x + c
```

в котором объекты `a`, `b`, `c`, `x` и `z` могут быть массивами, содержащими сотни числовых значений. Как видим, числовые массивы упрощают обозначения.

[11] Благодарю Дэвида Вандевурда за это замечание.

Кроме того, массивы значений обеспечивают высокую производительность обработки данных, поскольку классы обеспечивают оптимизацию, предотвращая создание временных объектов во время выполнения оператора. Эта оптимизация возможна благодаря тому, что у массивов значений нет псевдонимов. Иначе говоря, доступ к любому значению неконстантного массива значений осуществляется по уникальному пути. Это позволяет лучше оптимизировать операции с массивами, поскольку компилятору не приходится учитывать возможность доступа к данным по другому пути.

К тому же специальные интерфейсы и вспомогательные классы позволяют ограничивать работу подмножеством элементов массива значений и работать с многомерными массивами. Благодаря этому массивы значений позволяют реализовывать векторные и матричные вычисления и создавать соответствующие классы.

Проблемы, связанные с массивами значений

Классы массивов значений спроектированы не очень хорошо. Фактически никто и не пытался проверить работоспособность итоговой спецификации, потому что никто не считал себя ответственным за разработку этих классов. Люди, включившие массивы значений в стандартную библиотеку C++, вышли из комитета по стандартизации задолго до завершения стандарта. В результате массивы значений используются редко.

По указанным выше причинам и чтобы не увеличивать размер книги еще больше, описание класса `valarry` изложено во вспомогательной главе на веб-странице по адресу `http://www.cppstdlib.com`.

Глава 18

Параллельное программирование

Современные системные архитектуры обычно поддерживают одновременное выполнение нескольких задач и потоков. Если система к тому же оснащена несколькими процессорами, то применение нескольких потоков значительно уменьшает время выполнения программ.

Однако параллельная работа создает новые проблемы. В этом случае, вместо того чтобы выполнять инструкции одну за другой, система выполняет несколько инструкций одновременно. Это приводит к конкуренции за доступ к ресурсам, так что создание, чтение, запись и удаление не выполняются в ожидаемом порядке и могут привести к непредсказуемым результатам. Конкурентный доступ к данным из нескольких потоков легко может стать ночным кошмаром с такими проблемами, по сравнению с которыми взаимная блокировка, при которой потоки ждут друг друга, покажется мелкой неприятностью.

До принятия стандарта C++11 языковые средства C++ и стандартная библиотека C++ не поддерживали параллельность, хотя реализации могли обеспечивать некоторые гарантии. После принятия стандарта C++11 ситуация изменилась.

Ядро языка и библиотека теперь поддерживают параллельное программирование (см. раздел 4.5).

- Ядро языка определяет модель памяти, гарантирующую, что модификации двух разных объектов, используемых двумя разными потоками, не зависят друг от друга. Кроме того, появилось новое ключевое слово `thread_local` для определения переменных, принимающих значения, специфичные для потоков.
- Библиотека поддерживает запуск нескольких потоков, включая передачу аргументов, возвращаемые значения и передачу исключений через границы потоков, а также средства синхронизации нескольких потоков, так что существует возможность синхронизировать потоки управления и доступ к данным.

Библиотека обеспечивает эту поддержку на разных уровнях. Например, высокоуровневый интерфейс позволяет запускать поток, включая передачу аргументов и обработку результатов и исключений. Этот механизм основан на совокупности низкоуровневых интерфейсов, предназначенных для каждого из этих аспектов. С другой стороны, существуют низкоуровневые средства, такие как мьютексы и даже атомарные типы и операции (atomics), осуществляющие свободный доступ к памяти (relaxed memory order).

Настоящая глава посвящена именно этим средствам библиотеки. Отметим, что параллельность и библиотечные механизмы для ее поддержки описаны во многих книгах. По этой причине в главе излагаются лишь общие принципы и описываются типичные примеры, понятные среднестатистическому прикладному программисту, при этом основное внимание уделено высокоуровневым интерфейсам.

Описание конкретных подробностей, особенно сложных низкоуровневых проблем и средств их решения, можно найти в специальных книгах и статьях. В первую очередь

следует обратить внимание на основные принципы параллельного программирования, изложенные в книге Энтони Вильямса (Anthony Williams) *C++ Concurrency in Action* ([*Williams:C++Conc*]).

Э. Вильямс — один из ведущих специалистов по этой теме, и без его вклада данная глава не могла бы быть написана. Кроме своей книги, он предоставил в мое распоряжение первую стандартную библиотеку для параллельного программирования (см. [*JustThread*]), написал несколько статей и дал несколько ценных советов, которые помогли мне правильно, как я надеюсь, раскрыть эту тему. Кроме того, я бы хотел поблагодарить еще несколько экспертов в этой области, которые помогли мне написать главу: Ханса Бёма (Hans Boehm), Скотта Мейерса (Scott Meyers), Бартоша Милевски (Bartosz Milewski), Лоуренса Краула (Lawrence Crowl) и Питера Саммерленда (Peter Sommerlad).

Глава организована следующим образом.

- Сначала описываются разные способы запуска нескольких потоков, а после описания высоко- и низкоуровневых интерфейсов излагаются детали запуска потоков.
- Раздел 18.4 посвящен подробному обсуждению вопросов синхронизации. Основная проблема в этой теме — параллельный доступ к данным.
- В заключение рассматриваются разные функциональные возможности для синхронизации и параллельного доступа к данным.
  - Мьютексы и блокировки (раздел 18.5), включая `call_once()` (раздел 18.5.3).
  - Условные переменные (раздел 18.6).
  - Атомарность (раздел 18.7).

18.1. Высокоуровневый интерфейс: `async()` и `future<>`

Для новичков самым удобным способом запуска программы с несколькими потоками является высокоуровневый интерфейс стандартной библиотеки C++, предоставляемый функцией `std::async()` и классом `std::future<>`.

- Функция `async()` обеспечивает интерфейс для *вызываемого объекта* (см. раздел 4.4), по возможности выполняемого на фоне в качестве отдельного потока.
- Класс `future<>` позволяет ожидать завершения потока и предоставляет доступ к его результату: возвращаемому значению или исключению.

В данном разделе подробно описан высокоуровневый интерфейс и дано краткое введение в класс `std::shared_future<>`, который позволяет ожидать завершения потока и обрабатывать его результаты в нескольких местах.

18.1.1. Первый пример использования функции `async()` и класса `future<>`

Допустим, требуется вычислить сумму двух операндов, возвращаемых при двух вызовах функций. Обычно такое выражение записывается следующим образом:

```
func1() + func2()
```

Это значит, что обработка операндов выполняется последовательно. Эта программа сначала вызовет функцию `func1()`, а затем `func2()`, или наоборот (по правилам языка порядок остается неопределенным). В обоих случаях время вычислений складывается из продолжительности выполнения функций `func1()` и `func2()` и времени, которое занимает вычисление их суммы.

В настоящее время, когда многопроцессорные компьютеры стали общедоступными, все это можно сделать быстрее. Можно по крайней мере попытаться выполнить функции `func1()` и `func2()` параллельно, чтобы общая продолжительность их работы определялась максимальной продолжительностью выполнения функций `func1()` и `func2()`, а также времени вычисления их суммы.

Рассмотрим первую программу, решающую поставленную задачу:

```
// concurrency/async1.cpp

#include <future>
#include <thread>
#include <chrono>
#include <random>
#include <iostream>
#include <exception>
using namespace std;

int doSomething (char c)
{
   // генератор случайных чисел
   //(значение c используется для генерирования разных последовательностей)
   std::default_random_engine dre(c);
   std::uniform_int_distribution<int> id(10,1000);

   // цикл вывода символов после случайного момента времени
   for (int i=0; i<10; ++i) {
        this_thread::sleep_for(chrono::milliseconds(id(dre)));
        cout.put(c).flush();
   }
   return c;
}

int func1 ()
{
   return doSomething('.');
}

int func2 ()
{
   return doSomething('+');
}

int main()
{
   std::cout << "starting func1() in background"
             << " and func2() in foreground:" << std::endl;
```

```
    // начинаем асинхронное выполнение func1() (сейчас, позднее или никогда)
    std::future<int> result1(std::async(func1));
    int result2 = func2(); // вызываем func2() синхронно (здесь и сейчас)

    // выводим результат (ждем завершения функции func1() и добавляем ее
    // результат к переменной result2
    int result = result1.get() + result2;
    std::cout << "\nresult of func1()+func2(): " << result
              << std::endl;
}
```

Для визуализации происходящего мы моделируем сложные вычисления в функциях `func1()` и `func2()`, вызывая функцию `doSomething()`, которая время от времени выводит символ, передаваемый как аргумент[1], и в конце концов возвращает значение переданного символа как переменную типа `int`. "Время от времени" означает, что для определения интервала между выводами используется генератор случайных чисел, который функция `std::this_thread::sleep_for()` использует для задержки текущего потока (подробности, касающиеся генераторов случайных чисел, см. в разделе 17.1, а функция `sleep_for()` описана в разделе 18.3.7). Отметим, что для конструктора генератора случайных чисел необходимо *уникальное* начальное значение (здесь мы используем переданный символ `c`), которое гарантирует, что будут созданы разные последовательности случайных чисел.

Вместо вызова

```
int result = func1() + func2();
```

мы вызываем функции следующим образом:

```
std::future<int> result1(std::async(func1));
int result2 = func2();
int result = result1.get() + result2;
```

Итак, сначала мы *пытаемся* начать выполнение функции `func1()` в фоновом режиме, используя функцию `std::async()`, и присваиваем результат объекту класса `std::future` (фьючерсу. — *Примеч. ред.*).

```
std::future<int> result1(std::async(func1));
```

Здесь функция `async()` *пытается* немедленно начать выполнение переданной ей функции асинхронно в отдельном потоке. Таким образом, выполнение функции `func1()` теоретически начинается в этой точке без блокировки функции `main()`. Возвращаемый фьючерс необходим по двум причинам.

1. Он открывает доступ к будущему результату работы функциональной сущности, переданной функции `async()` как аргумент. Этим результатом может быть как возвращаемое значение, так и исключение. Класс `future` специализируется типом значения, возвращаемого функциональной сущностью, выполнение которой начинается в фоновом режиме. Если требуется просто выполнить какую-то работу в фоновом режиме, не возвращая никаких значений, следует использовать объект класса `std::future<void>`.
2. Необходимо гарантировать, что рано или поздно переданная функция будет вызвана. Обратите внимание на то, что функция `async()` *пытается* начать выполнение

[1] Вывод параллельными потоками возможен, но может привести к появлению промежуточных символов (см. раздел 4.5).

переданной функциональной сущности, а если этого не произойдет, то фьючерс позволит принудительно начать ее выполнение. Таким образом, фьючерс необходим, даже если нас не интересует результат работы функциональной сущности, выполняемой в фоновом режиме.

Для того чтобы обмениваться данными между точкой старта функциональной сущности и точкой возвращения фьючерса, они оба ссылаются на так называемое *общее (разделяемое) состояние* (*shared state*); см. раздел 18.3.

Разумеется, для объявления фьючерса можно и даже нужно использовать ключевое слово `auto` (продемонстрируем это явно):

```
auto result1(std::async(func1));
```

Затем мы начинаем выполнение функции `func2()` в фоновом режиме. Это обычный синхронный вызов функции, поэтому программа здесь блокируется.

```
int result2 = func2();
```

Таким образом, если выполнение функции `func1()` было успешно инициировано функцией `async()` и еще не закончилось, то функции `func1()` и `func2()` выполняются параллельно.

После этого мы вычисляем сумму. В этот момент нам нужен результат функции `func1()`. Для того чтобы получить его, вызываем функцию `get()` из фьючерса:

```
int result = result1.get() + result2;
```

При вызове функции `get()` могут произойти три события.

1. Если выполнение функции `func1()` было начато функцией `async()` в отдельном потоке и уже закончилось, то мы немедленно получим результат.
2. Если выполнение функции `func1()` было начато, но еще не закончилось, то функция `get()` блокирует поток и ждет, пока выполнение функции `func1()` будет закончено, чтобы получить результат.
3. Если выполнение функции `func1()` еще не начиналось, то она начнет выполняться принудительно как обычная синхронная функция, а функция `get()` заблокирует поток и будет ждать результата.

Эта схема очень важна. Она гарантирует, что программа будет работать в одно- или многопоточных средах, в которых по каким-то причинам функция `async()` не смогла запустить новый поток.

Вызов функции `async()` не гарантирует, что передаваемая функциональная сущность будет начата и закончена. Если поток доступен, ее выполнение будет начато, а если нет — например, если среда не поддерживает многопоточность или в ней больше нет доступных потоков, — то вызов будет отложен до момента, пока программист явно не укажет, что ему нужен результат (вызвав функцию `get()`), или просто захочет выполнить функцию (вызвав функцию `wait()`; см. раздел 18.1.1).

Таким образом, комбинация вызовов

```
std::future<int> result1(std::async(func1));
```

и

```
result1.get()
```

позволяет оптимизировать программу так, чтобы, по возможности, функция `func1()` выполнялась параллельно, пока выполняются следующие инструкции в главном потоке. Если ее невозможно выполнить параллельно, то она будет вызвана последовательно при вызове функции `get()`. Это значит, что в любом случае гарантируется, что после функции `get()` функция `func1()` будет вызвана асинхронно или синхронно.

Соответственно программа может выдать два вида результатов. Если функция `async()` смогла успешно начать выполнение функции `func1()`, то вывод может быть следующим:

```
starting func1() in background and func2() in foreground:
++..+++.++.+.+.....
result of func1()+func2(): 89
```

Если функция `async()` не смогла начать выполнение функции `func1()`, она будет выполнена после выполнения функции `func2()` при вызове функции `get()` и программа выдаст следующий результат:

```
starting func1() in background and func2() in foreground:
++++++++++.........
result of func1()+func2(): 89
```

Итак, из первого примера следует, что мы можем ускорить выполнение программы, модифицируя ее так, чтобы она использовала возможности параллельного выполнения, если базовая платформа поддерживает их, или работала, как обычно, в однопоточных средах. Для этого необходимо сделать следующее.

- Включить в программу директиву `#include <future>`.
- Передать функцию, которую можно выполнить параллельно как *вызываемый объект*, в виде аргумента функции `std::async()`.
- Присвоить результат объекту класса `future<`*ReturnType*`>`.
- Вызвать функцию `get()` из фьючерса, если нужен результат или выполнение начатой функции должно быть завершено.

Эти действия имеют смысл, только если не возникает состязание за данные (data race), означающее, что два потока одновременно используют одни и те же данные и приводят к непредсказуемым последствиям (см. раздел 18.4.1).

Если не вызвать функцию `get()`, то нет никакой гарантии, что функция `func1()` будет вообще когда-либо вызвана. Как указывалось ранее, если функция `async()` не может немедленно начать выполнение переданной функциональной сущности, то она *откладывает* это до того момента, пока функция `get()` (или `wait()`) запросит результаты ее работы. Без этого запроса прекращение функции `main()` приведет к завершению программы без вызова потока, выполняемого на фоне.

Программист должен гарантировать, что запрос на данные функции, которая запускается функцией `async()`, не поступит раньше, чем необходимо. Например, следующая оптимизация может быть совсем не желательной:

```
std::future<int> result1(std::async(func1));

int result = func2() + result1.get(); // может вызвать func2()
                                      // после завершения функции func1()
```

Поскольку порядок вычисления правой части второй инструкции не определен, функция `result1.get()` может быть вызвана до функции `func2()`, так что снова потребуется последовательная обработка.

Для того чтобы достичь наилучшего эффекта, необходимо обеспечить максимальный интервал между моментами вызовами функций `async()` и `get()`. В терминах работы [*N3194:Futures*] этот принцип можно сформулировать так: *рано вызывать и поздно возвращать результат*.

Если операция, передаваемая функции `async()`, ничего не возвращает, то функция `async()` возвращает объект типа `future<void>`, представляющий собой частичную специализацию класса `future<>`. В этом случае функция `get()` ничего не возвращает:

```
std::future<void> f(std::async(func)); // пытаемся выполнить
                                       // функцию func асинхронно
...
f.get(); // ожидаем завершения функции func (возвращает тип void)
```

В заключение отметим, что объект, переданный функции `async()`, может быть любым *вызываемым объектом* (callable object): функцией, функцией-членом, функциональным объектом или лямбда-функцией (см. раздел 4.4). Таким образом, любую функциональную сущность, которая должна выполняться в своем потоке, можно передать как лямбда-функцию (см. раздел 3.1.10):

```
std::async([]{ ... }) // пытаемся выполнить ... асинхронно
```

Использование стратегий запуска

Функции `async()` можно запретить отсрочку запуска переданной функциональной сущности, явно передав *стратегию запуска* (launch policy)[2], указав функции `async()`, что она должна явно запустить функциональную сущность асинхронно в момент ее вызова:

```
// вынуждаем немедленное асинхронное выполнение функции func1() или
// генерируем исключение std::system_error
std::future<long> result1= std::async(std::launch::async, func1);
```

Если асинхронный вызов в этой точке невозможен, то программа сгенерирует исключение `std::system_error` (см. раздел 4.3.1) с кодом ошибки `resource_unavailable_try_again`, что эквивалентно сообщению об ошибке `errno EAGAIN` в стандарте POSIX (см. раздел 4.3.2).

Стратегия запуска `async` позволяет не вызывать функцию `get()`, потому что по завершении интервала существования возвращаемого фьючерса программа будет ожидать выполнения функции `func1()`, чтобы закончить работу. Таким образом, если функция `get()` не вызывается, то выход фьючерса из области видимости (в данном случае она заканчивается в конце функции `main()`) будет отложен до момента, пока будет завершена фоновая задача. Тем не менее вызов функции `get()` до завершения программы делает ее работу более понятной.

[2] Стратегия запуска представляет собой *перечисление с ограниченной областью видимости* (scoped enumeration), поэтому значения перечисления должны иметь квалификатор `std::launch` или `launch` (см. раздел 3.1.13).

Если результат функции `std::async(std::launch::async,...)` нигде не присваивается, то вызывающая сторона будет заблокирована, пока не будет завершена переданная функциональная сущность. По существу, это является синхронным вызовом[3].

Аналогично можно вынудить отложенное выполнение, передав функции `async()` стратегию запуска `std::launch:deferred`. В следующем фрагменте мы откладываем выполнение функции `func1()`, пока функция `get()` не будет вызвана из объекта `f`:

```
std::future<...> f(std::async(std::launch::deferred,
                              func1)); // откладываем выполнение функции func1,
                                       // пока не будет вызвана функция get()
```

Этот код не гарантирует, что функция `func1()` никогда не будет вызвана без помощи функции `get()` (или `wait()`).

Данная стратегия позволяет программе выполнять *отложенные вычисления* (lazy evaluation). Рассмотрим пример[4]:

```
auto f1 = std::async( std::launch::deferred, task1 );
auto f2 = std::async( std::launch::deferred, task2 );
...
auto val = thisOrThatIsTheCase() ? f1.get() : f2.get();
```

Кроме того, явное указание стратегии `deferred` может помочь симулировать поведение функции `async()` в однопоточной среде или упростить отладку (если режим состязания создает проблему).

Работа с исключениями

До сих пор мы обсуждали только случай, когда задачи в потоках и в фоновом режиме выполнялись успешно. А что произойдет, если возникнет исключение?

Оказывается, ничего особенного; функция `get()` для фьючерсов также обрабатывает исключения. Фактически, когда вызывается функция `get()`, а выполнение фоновой операции было прекращено из-за исключения, которое не было обработано в потоке, это исключение передается вновь. В результате для работы с исключениями, порожденными фоновыми операциями, с функцией `get()` надо работать так, как будто операция была вызвана синхронно.

Для примера запустим фоновую задачу с бесконечным циклом, в котором выделяется память для вставки нового элемента в список[5]:

```
// concurrency/async2.cpp

#include <future>
#include <list>
```

[3] В комитете по стандартизации возникла дискуссия о том, как интерпретировать текущую формулировку, если результат работы функции `async()` не используется. Она стала результатом бурных обсуждений и должна учитываться во всех реализациях.

[4] Благодарю Лоуренса Кроула за указанное замечание и пример.

[5] Конечно, попытка работать с памятью, пока не возникнет исключение, — плохая практика, которая подвергает операционную систему опасности. Это следует иметь в виду перед тем, как компилировать данный пример.

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

void task1()
{
   // бесконечная вставка и выделение памяти
   // - рано или поздно это приведет к исключению
   // - ПРЕДУПРЕЖДЕНИЕ: это плохая практика
   list<int> v;
   while (true) {
      for (int i=0; i<1000000; ++i) {
          v.push_back(i);
      }
      cout.put('.').flush();
   }
}

int main()
{
   cout << "starting 2 tasks" << endl;
   cout << "- task1: process endless loop of memory consumption" << endl;
   cout << "- task2: wait for <return> and then for task1" << endl;

   auto f1 = async(task1); // запускаем task1() асинхронно
                           // (сейчас, позднее или никогда)

   cin.get();              // вводим символ (как getchar())

   cout << "\nwait for the end of task1: " << endl;
   try {
      f1.get(); // ожидаем завершения task1() (или исключения)
   }
   catch (const exception& e) {
      cerr << "EXCEPTION: " << e.what() << endl;
   }
}
```

Рано или поздно бесконечный цикл вызовет исключение (вероятно, `bad_alloc`; см. раздел 4.3.1), которое прервет выполнение потока, потому что оно не было перехвачено. Фьючерс будет сохранять это состояние, пока не будет вызвана функция `get()`. После вызова функции `get()` исключение будет передано далее в функцию `main()`.

Теперь интерфейс функции `async()` и класса `future` можно описать следующим образом. Функция `async()` дает среде программирования шанс начать параллельный процесс, результат которого будет использован позднее (когда будет вызвана функция `get()`). Иначе говоря, если есть независимая функциональная сущность `f`, можно воспользоваться возможностью ее параллельного выполнения, передав `f` функции `async()` в момент, когда нужно вызвать функцию `f`, и заменив выражение, в котором требуется результат работы функции `f`, вызовом функции `get()` из фьючерса. Таким образом, результат будет тем же самым, но производительность увеличивается, потому что функция `f` может быть выполнена параллельно еще до того, как понадобится ее результат.

Ожидание и опрос

Функцию `get()` для фьючерса можно вызвать только один раз. После вызова функции `get()` фьючерс переходит в некорректное состояние, которое можно проверить, только вызвав функцию `valid()` для фьючерса. Вызов любой другой функции, кроме деструктора, приводит к неопределенным последствиям (см. раздел 18.3.2).

Кроме того, фьючерсы обеспечивают интерфейс для ожидания завершения фоновой операции без обработки ее результата. Этот интерфейс можно вызывать несколько раз. Его можно также комбинировать с заданием продолжительности выполнения или ожидаемого момента времени для ограничения времени ожидания.

Функция `wait()` принудительно начинает выполнение потока, который представляет фьючерс, и ожидает прекращения фоновой операции:

```
std::future<...> f(std::async(func)); // пытаемся вызвать функцию func асинхронно
...
f.wait(); // ожидаем завершения функции func (может начинать фоновую задачу)
```

У фьючерсов есть еще две функции `wait()`, но они *не инициируют* запуск потока, если он еще не начинался.

1. Функция `wait_for()` может ждать завершения асинхронной операции ограниченное время в течение заданного периода времени.

   ```
   std::future<...> f(std::async(func)); // пытаемся вызвать func асинхронно
   ...
   f.wait_for(std::chrono::seconds(10)); // ждем функцию func 10 секунд
   ```

2. Функция `wait_until()` ожидает, когда наступит конкретный момент времени.

   ```
   std::future<...> f(std::async(func)); // // пытаемся вызвать func асинхронно
   ...
   f.wait_until(std::system_clock::now()+std::chrono::minutes(1));
   ```

Функции `wait_for()` и `wait_until()` возвращают один из следующих объектов.

- Объект класса `std::future_status::deferred`, если функция `async()` отложила операцию, а функция `wait()` или `get()` еще не заставила начать ее выполнение (обе функции в этом случае немедленно возвращают результат).
- Объект `std::future_status::timeout`, если операция была начата асинхронно, но еще не завершена (период ожидания еще не истек).
- Объект `std::future_status::ready`, если операция завершена.

Функции `wait_for()` и `wait_until()` особенно полезны для проведения *спекулятивных вычислений* (speculative execution). Например, рассмотрим сценарий, в котором в определенный момент времени необходимо получить точный результат вычислений[6]:

```
int quickComputation();       // вычисляет результат "быстро, но неточно"
int accurateComputation();    // вычисляет результат "точно, но медленно"

std::future<int> f;           // объявлена во внешнем файле, потому что
                              // время существования функции
```

[6] Благодарю Лоуренса Кроула за указанное замечание и пример.

```
                                // accurateComputation() возможно превысило
                                // время существования функции bestResultInTime()

int bestResultInTime()
{
   // определяем временной интервал для получения ответа:
   auto tp = std::chrono::system_clock::now() + std::chrono::minutes(1);

   // начинаем быстрые и точные вычисления
   f = std::async (std::launch::async, accurateComputation);
   int guess = quickComputation();

   // проводим точные вычисления на протяжении оставшегося периода времени
   std::future_status s = f.wait_until(tp);

   // возвращаем наилучший из имеющихся результатов
   if (s == std::future_status::ready) {
       return f.get();
   }
   else {
       return guess; // продолжаем accurateComputation()
   }
}
```

Фьючерс `f` не может быть локальным объектом в функции `bestResultInTime()`, потому что, если временной интервал слишком короткий, чтобы функция `accurateComputation()` успела завершить свое выполнение, деструктор фьючерса может заблокировать поток, пока асинхронная задача не будет завершена.

Передав интервал нулевой длины или нулевой момент времени, можно просто спровоцировать опрос, чтобы увидеть, будет ли начато выполнение фоновой задачи и/или выполняется ли она:

```
future<...> f(async(task)); // пытаемся выполнить задачу асинхронно
...
// что-то делаем, пока задача не завершится (это может никогда не случиться!)
while (f.wait_for(chrono::seconds(0)) != future_status::ready) {
   ...
}
```

Тем не менее такой цикл может никогда не закончиться, потому что, например, в однопоточных средах вызов может быть отложен до момента, когда будет вызвана функция `get()`. Следовательно, необходимо либо вызвать функцию `async()` со стратегией `std::launch::async`, переданной как первый аргумент, либо явно проверить, возвращает ли функция `wait_for()` объект типа `std::future_status::deferred`:

```
future<...> f(async(task)); // пытаемся выполнить задачу асинхронно
...
   // проверяем, была ли отложена задача.
   if (f.wait_for(chrono::seconds(0)) != future_status::deferred) {

       // делаем что-то, пока задача не будет завершена
       while (f.wait_for(chrono::seconds(0) != future_status::ready)) {
       ...
```

```
        }
    }
  ...
auto r = f.get(); // вынуждаем выполнение задачи и ожидаем результата
                  // (или исключения)
```

Другая причина бесконечного цикла может заключаться в том, что поток, выполняющий цикл, занимает процессор и другие потоки не получают возможности перевести фьючерс в состояние готовности. Это может резко снизить быстродействие программы. Проще всего исправить ситуацию, вызвав функцию `yield()` (см. раздел 18.3.7) в цикле, и/или сделать паузу на короткое время:

```
std::this_thread::yield(); // подсказка изменить расписание
                           // для следующего потока
```

Подробное описание интервалов и моментов времени, которые можно передавать как аргументы функциям `wait_for()` и `wait_until()`, см. в разделе 5.7. Отметим, что функции `wait_for()` и `wait_until()` обычно отличаются друг от друга способом корректировки системного времени (см. раздел 5.7.5).

18.1.2. Пример ожидания двух задач

Описанные выше возможности демонстрирует следующий пример:

```
// concurrency/async3.cpp

#include <future>
#include <thread>
#include <chrono>
#include <random>
#include <iostream>
#include <exception>
using namespace std;

void doSomething (char c)
{
   // генератор случайных чисел (используем переменную с как начальное значение,
   // чтобы получать разные последовательности)
   default_random_engine dre(c);
   uniform_int_distribution<int> id(10,1000);

   // цикл вывода символов через случайные интервалы времени
   for (int i=0; i<10; ++i) {
        this_thread::sleep_for(chrono::milliseconds(id(dre)));
        cout.put(c).flush();
   }
}

int main()
{
   cout << "starting 2 operations asynchronously" << endl;

   // начинаем два цикла на фоне для вывода символа . или +
```

```
    auto f1 = async([]{ doSomething('.'); });
    auto f2 = async([]{ doSomething('+'); });

    // если хотя бы одна из фоновых задач выполняется
    if (f1.wait_for(chrono::seconds(0)) != future_status::deferred ||
        f2.wait_for(chrono::seconds(0)) != future_status::deferred) {
       // опрашиваем, не выполнен ли хотя бы один цикл
       while (f1.wait_for(chrono::seconds(0)) != future_status::ready &&
              f2.wait_for(chrono::seconds(0)) != future_status::ready) {
          ...;
          this_thread::yield(); // подсказка изменить расписание
                                // для следующего потока
       }
    }
    cout.put('\n').flush();

    // ожидаем, пока не будут завершены все циклы, и обрабатываем все исключения
    try {
        f1.get();
        f2.get();
    }
    catch (const exception& e) {
        cout << "\nEXCEPTION: " << e.what() << endl;
    }
    cout << "\ndone" << endl;
}
```

И снова здесь используется операция `doSomething()`, которая время от времени выводит символ, передаваемый как аргумент (см. раздел 18.1.1).

Теперь с помощью функции `async()` мы дважды запускаем функцию `doSomething()` в фоновом режиме, выводя два разных символа, используя разные задержки, сгенерированные соответствующими последовательностями случайных чисел.

```
auto f1 = std::async([]{ doSomething('.'); });
auto f2 = std::async([]{ doSomething('+'); });
```

В результате в многопоточных средах можно одновременно выполнять две операции, которые время от времени выводят разные символы.

Затем мы выполняем опрос, чтобы увидеть, не завершилась ли одна из операций[7].

```
while (f1.wait_for(chrono::seconds(0)) != future_status::ready &&
       f2.wait_for(chrono::seconds(0)) != future_status::ready) {
   ...
   this_thread::yield(); // подсказка изменить расписание
                         // для следующего потока


}
```

[7] Если бы в циклах мы не выполняли полезные операции, это было бы простой задержкой, т.е. проблему лучше было бы решать с помощью условных переменных (см. раздел 18.6.1).

Но, поскольку этот цикл никогда не завершается, если ни одна из задач не была выполнена в фоновом режиме при вызове функции `async()`, сначала необходимо проверить, не была ли отложена одна из операций.

```
if (f1.wait_for(chrono::seconds(0)) != future_status::deferred ||
    f2.wait_for(chrono::seconds(0)) != future_status::deferred) {
    ...
}
```

В качестве альтернативы можно было бы вызвать функцию `async()` со стратегией запуска `std::launch:async`.

Если хотя бы одна из фоновых операций была завершена или не одна из них еще не начинала выполняться, выводим символ перехода на новую строку и ожидаем завершения обоих циклов.

```
f1.get();


f2.get();
```

Для обработки исключения, которое может возникнуть, используется функция `get()`.

В многопоточной среде программа может вывести, например, следующий результат:

```
starting 2 operations asynchronously
++.++..+.+..++.+.+
..
done
```

Порядок следования всех трех символов (`.`, `+` и *символа перехода на новую строку*) не гарантируется. Обычно первой выводится точка, поэтому это делает первая инструкция (таким образом, ее выполнение начинается немного раньше), но символ `+` также может выводиться первым. Символы `.` и `+` могут быть перемешаны, но это тоже не гарантируется. Если удалить инструкцию с функцией `sleep_for()`, которая порождает задержку между каждым выводом символа, то первый цикл выполняется до первого переключения на другой поток и вывод может выглядеть следующим образом:

```
starting 2 operations asynchronously
..........
++++++++++
done
```

Такой вывод может также возникнуть в среде, не поддерживающей многопоточность, потому что в этом случае оба вызова функции `doSomething()` будут выполнены синхронно с вызовами функции `get()`.

Кроме того, не понятно, когда будет выведен символ перехода на новую строку. Он может выводиться перед всеми остальными символами, если выполнение обеих фоновых задач отложено до вызова функции `get()`. В этом случае отложенные задачи будут выполнены одна за другой:

```
starting 2 operations asynchronously
..........++++++++++
done
```

Мы знаем лишь то, что символ перехода на новую строку не будет выведен до того, как будет закончен один из циклов. Не гарантируется даже, что этот символ перехода на новую строку будет следовать непосредственно за последним символом одной из последовательностей, потому что для записи конца одного из циклов во фьючерс и оценки записанного состояния требуется определенное время (этот процесс не протекает в реальном времени). По этой причине результат может содержать несколько символов +, записанных между последней точкой и символом перехода на новую строку:

```
starting 2 operations asynchronously
.+..+..+..+.+..++
+++
done
```

Передача аргументов

Предыдущий пример демонстрирует один из способов передачи аргументов фоновой задаче: здесь просто используется лямбда-функция (см. раздел 3.1.10), которая вызывает фоновую функциональную сущность:

```
auto f1 = std::async([]{ doSomething('.'); });
```

Разумеется, можно также передавать аргументы, существовавшие до вызова `async()`. Как обычно, их можно передать по значению или по ссылке.

```
char c = '@';
auto f = std::async([=]{ // =: открываем доступ к объектам
                         // в области видимости по значению
                         doSomething(c); // передаем копию c в doSomething()
});
```

Определяя *захват* как `[=]`, можно передать лямбда-функции *копию* `c` и все видимые объекты, поэтому в лямбда-функции можно передать эту копию `c` в функцию `doSomething()`.

Однако существуют другие способы передачи аргументов в функцию `async()`, так как `async()` обеспечивает обычный интерфейс *вызываемых объектов* (см. раздел 4.4). Например, если передать указатель на функцию как первый аргумент функции `async()`, то можно передать несколько дополнительных аргументов, которые являются параметрами вызываемой функциональной сущности.

```
char c = '@';
auto f = std::async(doSomething,c); // вызываем doSomething(c) асинхронно
```

Аргументы можно передавать по ссылке, но в этом случае существует риск, что передаваемые значения станут некорректными еще до старта фоновой задачи. Это относится как к обычным, так и лямбда-функциям, вызываемым непосредственно.

```
char c = '@';
auto f = std::async([&]{ doSomething(c); });   // опасно!

char c = '@';
auto f = std::async(doSomething,std::ref(c));  // опасно!
```

Если вы контролируете время существования передаваемого аргумента так, что оно превышает время решения фоновой задачи, то это можно делать. Рассмотрим пример:

```
void doSomething (const char& c);              // передаем символ по ссылке
...
char c = '@';
auto f = std::async([&]{ doSomething(c); }); // передаем c по ссылке
...
f.get();  // время существования переменной c должно достигать этой точки
```

Впрочем, будьте осторожны: если вы передаете аргументы по ссылке, чтобы модифицировать их в отдельном потоке, то можете легко вызвать непредсказуемые последствия. Рассмотрим следующий пример, в котором после вывода символа в фоновом режиме мы переключаем символ, подлежащий выводу:

```
void doSomething (const char& c);              // передаем символ по ссылке
...
char c = '@';
auto f = std::async([&]{ doSomething(c); });  // передаем переменную c по ссылке
...
c = '_'; // переключаем вывод функции doSomething() на символы подчеркивания,
         // если она все еще выполняется
f.get(); // время существования переменной c должно достигать этой точки
```

Во-первых, порядок доступа к переменной `c` здесь и в функции `doSomething()` не определен. Таким образом, переключение символа, подлежащего выводу, может произойти до цикла, внутри цикла или после цикла. Что еще хуже, поскольку мы модифицируем переменную `c` в одном потоке, а считывает ее другой поток, — возникает несинхронизированный конкурентный доступ к одному и тому же объекту (так называемое *состязание за данные* (data race), см. раздел 18.4.1). Это порождает непредсказуемые последствия, если не защитить конкурентный доступ с помощью мьютексов (см. раздел 18.5) или атомарных операций (см. раздел 18.7).

Итак, если вы собираетесь использовать функцию `async()`, то должны передавать ***все*** объекты, необходимые для работы функциональной сущности, только ***по значению***, чтобы функция `async()` работала исключительно с ***локальными копиями***. Если копирование представляет собой слишком затратную операцию, то убедитесь, что объекты передаются по константной ссылке, а ключевое слово `mutable` не используется.

В любом другом случае обращайтесь к разделу 18.4 и убедитесь, что вы правильно понимаете последствия своих действий.

Можно также передать функции `async()` указатель на функцию-член. В этом случае первый аргумент после функции-члена должен быть ссылкой или указателем на объект, из которого вызывается функция-член.

```
class X
{
   public:
      void mem (int num);
      ...
};
...
X x;
auto a = std::async(&X::mem, x, 42); // пытаемся вызвать x.mem(42) асинхронно
...
```

18.1.3. Разделяемые фьючерсы

Как было показано, класс `std::future` позволяет обрабатывать будущий результат параллельных вычислений. Однако этот результат можно обработать только один раз. Второй вызов функции `get()` порождает непредвиденные последствия (в соответствии с принципами стандартной библиотеки C++ желательно, но не обязательно, чтобы реализации генерировали исключение `std::future_error`).

Однако иногда целесообразно несколько раз обрабатывать результат параллельных вычислений, особенно если его обрабатывают несколько других потоков. Для этой цели стандартная библиотека C++ предоставляет класс `std::shared_future`, который допускает несколько вызовов функции `get()` и возвращает один и тот же результат или генерирует одно и то же исключение.

Рассмотрим следующий пример:

```
// concurrency/sharedfuture1.cpp

#include <future>
#include <thread>
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;

int queryNumber ()
{
   // считываем число
   cout << "read number: ";
   int num;
   cin >> num;

   // генерируем исключение, если ничего не введено
   if (!cin) {
       throw runtime_error("no number read");
   }

   return num;
}

void doSomething (char c, shared_future<int> f)
{
   try {
      // получаем количество символов, подлежащих выводу
      int num = f.get(); // get result of queryNumber()

      for (int i=0; i<num; ++i) {
           this_thread::sleep_for(chrono::milliseconds(100));
           cout.put(c).flush();
      }
   }
   catch (const exception& e) {
      cerr << "EXCEPTION in thread " << this_thread::get_id()
           << ": " << e.what() << endl;
```

```
    }
}

int main()
{
    try {
        // запускаем поток для запроса числа
        shared_future<int> f = async(queryNumber);

        // запускаем три потока, каждый из которых обрабатывает это число в цикле
        auto f1 = async(launch::async,doSomething,'.',f);
        auto f2 = async(launch::async,doSomething,'+',f);
        auto f3 = async(launch::async,doSomething,'*',f);

        // ждем завершения всех трех циклов
        f1.get();
        f2.get();
        f3.get();
    }
    catch (const exception& e) {
        cout << "\nEXCEPTION: " << e.what() << endl;
    }
    cout << "\ndone" << endl;
}
```

В этом примере один поток вызывает функцию `queryNumber()`, чтобы получить целочисленное значение, которое затем используется другими, уже работающими потоками. Для выполнения этого задания результат функции `std::async()`, которая запускает запрашивающий поток, присваивается объекту `shared_future`, специализированному типом возвращаемого значения:

```
shared_future<int> f = async(queryNumber);
```

Таким образом, разделяемый фьючерс можно инициализировать обычным фьючерсом, который переходит в состояние разделяемого фьючерса. Для того чтобы получить возможность использовать в этом объявлении ключевое слово `auto`, можно в качестве альтернативы использовать функцию-член `share()`:

```
auto f = async(queryNumber).share();
```

Все разделяемые фьючерсы имеют *общее состояние* (shared state), которое создается функцией `async()` для хранения результата работы передаваемой функциональной сущности (и самой этой сущности, если ее выполнение было отложено).

Разделяемый фьючерс затем передается другим потокам, начиная выполнение функции `doSomething()`, у которой вторым аргументом является разделяемый фьючерс:

```
auto f1 = async(launch::async,doSomething,'.',f);
auto f2 = async(launch::async,doSomething,'+',f);
auto f3 = async(launch::async,doSomething,'*',f);
```

В каждом вызове функции `doSomething()` мы ожидаем и обрабатываем результат работы функции `queryNumber()`, вызывая функцию `get()` из передаваемого разделяемого фьючерса.

```
void doSomething (char c, shared_future<int> f)
{
   try {
       int num = f.get(); // get result of queryNumber()
       ...
   }
   catch (const exception& e) {
       cerr << "EXCEPTION in thread " << this_thread::get_id()
            << ": " << e.what() << endl;
   }
}
```

Если не удается прочитать никакого целочисленного значения, функция `queryNumber()` генерирует исключение. В этом случае каждый вызов функции `doSomething()` будет получать это исключение с помощью инструкции `f.get()`, так что оно будет обработано.

Таким образом, после ввода числа 5 результат может быть следующим:

```
read number: 5
*+.*+.*.+*+.*.+
done
```

Если же ввести символ `'x'`, результат будет иным:

```
read number: x
EXCEPTION in thread 3: no number read
EXCEPTION in thread 4: no number read
EXCEPTION in thread 2: no number read
done
```

Отметим, что порядок вывода в потоках и значения их идентификаторов не определены (идентификаторы потоков описываются в разделах 18.2.1).

Кроме того, между определениями функции `get()` в классах `future` и `shared_future` есть небольшая разница.

- В классе `future<>` функция `get()` объявляется следующим образом (`T` — тип возвращаемого значения):

  ```
  T future<T>::get();          // общая функция get()
  T& future<T&>::get();        // специализация для ссылок
  void future<void>::get();    // специализация для void
  ```

  где первая версия возвращает перемещаемый результат или копию результата.

- В классе `shared_future<>` функция `get()` определяется следующим образом:

  ```
  const T& shared_future<T>::get();   // общая функция get()
  T& shared_future<T&>::get();        // специализация для ссылок
  void shared_future<void>::get();    // специализация для void
  ```

  где первая версия возвращает ссылку на значение результата, хранящееся в *общем состоянии*.

Как указано в [*N3194:Futures*]:

"Версия функции `get()` без параметров оптимизирована для перемещения (например, `std::vector<int> v = f.get()`). ... Версия функции `get()` для константной ссылки оптимизирована для доступа (например, `int i = f.get()[3]`)".

Такая схема создает риск, связанный с продолжительностью существования фьючерса или состязанием за данные при модификации возвращаемых значений (см. раздел 18.3.3). Разделяемый фьючерс можно также передать по ссылке (иначе говоря, объявить его как ссылку и передать с помощью функции `std::ref()`):

```
void doSomething (char c, const shared_future<int>& f)
auto f1 = async(launch::async,doSomething,'.',std::ref(f));
```

Теперь вместо использования нескольких разделяемых фьючерсных объектов, разделяющих *общее состояние*, можно использовать один разделяемый фьючерсный объект для выполнения нескольких функций `get()` (по одной в каждом потоке). Однако этот подход более рискованный. Программист обязан гарантировать, что продолжительность существования объекта `f` (да, именно `f`, а не *общего состояния*, на которое он ссылается) будет не меньше, чем у запускаемого потока. Кроме того, функции-члены разделяемых фьючерсов не синхронизируются друг с другом, хотя *общее состояние* синхронизируется.

Итак, если вы делаете нечто большее, чем простой ввод данных, то, возможно, вам потребуются внешние механизмы синхронизации (см. раздел 18.4), чтобы избежать *состязания за данные*, которое может вызвать непредвиденные последствия. Как сказал Лоуренс Кроул (Lowrence Crowl), один из авторов параллельной библиотеки, в одном из частных разговоров: "Если код остается четко скоординированным, передача по ссылке работает прекрасно. Если же цель и ограничения кода не вполне понятны, то лучше использовать передачу по значению. Копирование разделяемого фьючерса — затратная операция, но не настолько затратная, чтобы оправдать существование скрытых ошибок в крупной системе".

Дальнейшие подробности, касающиеся класса `shared_future`, изложены в разделе 18.3.3.

18.2. Низкоуровневый интерфейс: потоки и обещания

Помимо высокоуровневого интерфейса функции `async()` и (разделяемых) фьючерсов, в стандартной библиотеке C++ есть низкоуровневый интерфейс для запуска потоков и работы с ними.

18.2.1. Класс `std::thread`

Для запуска потока достаточно объявить объект класса `std::thread` и передать ему задачу, которую требуется решить, как начальный аргумент, а затем либо ожидать, пока поток закончит работу, либо *отсоединить* его.

```
void doSomething();
std::thread t(doSomething); // начинаем doSomething() в фоновом режиме
...
t.join(); // ожидаем, когда t закончит работу
          // (блокируем поток, пока не закончится
          // выполнение функции doSomething())
```

Как и при работе с функцией `async()`, потоку можно передавать все, что является *вызываемым объектом* (функцию, функцию-член, функциональный объект, лямбда-функцию; см. раздел 4.4), вместе с возможными аргументами. Однако снова следует отметить, что, несмотря на то, что мы знаем, что делаем, мы должны передать *все* объекты, необходимые для выполнения передаваемой функциональной сущности, *по значению*, чтобы объект класса `thread` использовал только *локальные копии* (проблемы, которые могут возникнуть, если этого не сделать, описаны в разделе 18.4).

Кроме того, это низкоуровневый интерфейс, значит, интерес представляет, что он делает такого, чего *не может* функция `async()` (см. раздел 18.1).

- Класс `thread` не имеет стратегии запуска. Стандартная библиотека C++ всегда пытается начать выполнение передаваемой функциональной сущности в новом потоке. Если это невозможно, она генерирует исключение `std::system_error` (см. раздел 4.3.1) с кодом ошибки `resource_unavailable_try_again` (см. раздел 4.3.2).
- В этом классе не существует интерфейса для обработки результата или вывода потока. Можно лишь получить уникальный идентификатор потока (см. раздел 18.2.1).
- Если возникает исключение, не перехваченное в потоке, программа немедленно выполняет аварийную остановку, вызывая функцию `std::terminate()` (см. раздел 5.8.2). Для того чтобы передать исключение во внешний контекст, следует использовать объекты типа `exception_ptr` (см. раздел 4.3.3).
- Как вызывающая сторона, вы должны объявить, хотите ли вы ожидать конца выполнения потока (вызвав функцию `join()`) или желаете *отсоединиться* от потока, запущенного в фоновом режиме, никак им не управляя (вызвав функцию `detach()`). Если этого не сделать до завершения существования объекта потока или если к потоку будет применена операция перемещающего присваивания, программа выполнит аварийную остановку, вызвав функцию `std::terminate()` (см. раздел 5.8.2).
- Если потоки выполняются в фоновом режиме и функция `main()` завершает работу, все потоки аварийно останавливаются.

Рассмотрим первый вариант:

```
// concurrency/thread1.cpp

#include <thread>
#include <chrono>
#include <random>
#include <iostream>
#include <exception>
using namespace std;

void doSomething (int num, char c)
{
   try {
      // генератор случайных чисел (используем с как начальное значение,
      // чтобы получать разные последовательности)
      default_random_engine dre(42*c);
      uniform_int_distribution<int> id(10,1000);
      for (int i=0; i<num; ++i) {
           this_thread::sleep_for(chrono::milliseconds(id(dre)));
           cout.put(c).flush();
```

```
            ...
        }
    }
    // убеждаемся, что поток не оставил исключений
    // и прекращаем выполнение программы
    catch (const exception& e) {
        cerr << "THREAD-EXCEPTION (thread "
             << this_thread::get_id() << "): " << e.what() << endl;
    }
    catch (...) {
        cerr << "THREAD-EXCEPTION (thread "
             << this_thread::get_id() << ")" << endl;
    }
}

int main()
{
    try {
        thread t1(doSomething,5,'.'); // выводим пять точек в отдельном потоке
        cout << "- started fg thread " << t1.get_id() << endl;

      // выводим другие символы в других фоновых потоках
      for (int i=0; i<5; ++i) {
            thread t(doSomething,10,'a'+i); // выводим 10 символов
                                            // в отдельном потоке
            cout << "- detach started bg thread " << t.get_id() << endl;
            t.detach(); // отсоединяем поток в фоновый режим
      }

      cin.get(); // ожидаем любого ввода (возвращения)
      cout << "- join fg thread " << t1.get_id() << endl;
      t1.join(); // wait for t1 to finish
    }
    catch (const exception& e) {
        cerr << "EXCEPTION: " << e.what() << endl;
    }
}
```

Здесь в функции `main()` мы запускаем несколько потоков, выполняющих функцию `doSomething()`. Функции `main()` и `doSomething()` содержат разделы `try-catch` по следующим причинам.

- В функции `main()` создание потока может сгенерировать исключение `std::system_error` (см. раздел 4.3.1) с кодом ошибки `resource_unavailable_try_again`.
- В функции `doSomething()`, начинающейся как `std::thread`, любое неперехваченное исключение может вызвать аварийную остановку программы.

Мы ожидаем завершения первого потока, запущенного в функции `main()`.

```
thread t1(doSomething,5,'.'); // выводим пять точек в отдельном потоке
...
t1.join();                    // ожидаем, пока завершится поток t1
```

Остальные потоки отсоединяются после запуска, поэтому могут выполняться до конца функции `main()`:

```
for (int i=0; i<5; ++i) {
     thread t(doSomething,10,'a'+i); // выводим 10 символов в отдельном потоке
     t.detach();                     // отсоединяем поток в фоновый режим
}
```

Вследствие этого программа может немедленно прекратить работу всех фоновых потоков по завершении функции `main()`, что может произойти, например, из-за функции `cin.get()`, если возможен ввод, *и* из-за функции `t1.join()`, если в качестве последнего символа в потоке, выполняющем вызов `doSomething(5,'.')`, записана пятая точка. Поскольку ожидание ввода и вывод точек выполняются параллельно, не имеет значения, какая из этих операций будет выполнена первой.

Например, программа может вывести следующие строки, если пользователь нажмет клавишу <Enter> после вывода второй точки.

```
- started fg thread 1
- detach started bg thread 2
- detach started bg thread 3
- detach started bg thread 4
- detach started bg thread 5
- detach started bg thread 6
ecad.dbcebabd.a
- join fg thread 1
b.ceade.bbcadbe.
```

Отсоединение потоков

Отсоединенные потоки легко могут стать проблемой, если они используют нелокальные ресурсы. Проблема заключается в том, что программист теряет управление отсоединенным потоком и не имеет простого способа определить, выполняется он до сих пор или нет. Таким образом, следует убедиться, что отсоединенные потоки не имеют доступа ни к каким объектам после того, как время их существования закончилось. По этой причине передача потоку параметров и объектов всегда является рискованной. Категорически рекомендуется использовать передачу аргументов по значению.

Однако проблема со временем существования относится и к глобальным, и к статическим объектам, потому что пока программа существует, отсоединенный поток попрежнему выполняется, т.е. он может иметь доступ к глобальным или статическим объектам, которые уничтожаются или уже уничтожены. К сожалению, это может вызвать непредвиденные последствия[8].

Итак, при работе с отсоединенными потоками следует учитывать следующие обстоятельства.

- Желательно, чтобы отсоединенные потоки работали только с локальными копиями.
- Если отсоединенный поток использует глобальный или статический объект, необходимо выполнить одно из следующих действий.

[8] Благодарю Ганса Боэма (Hans Boehm) и Энтони Уильямса (Anthony Williams) за то, что они указали на эту проблему.

- Гарантировать, что эти глобальные или статические объекты не уничтожаются до тех пор, пока не будут завершены все отсоединенные потоки, имеющие к ним доступ (или перестанут к ним обращаться). Для этого можно использовать условные переменные (см. раздел 18.6), которые используются отсоединенными потоками для сигнализации о своем завершении. Прежде чем выйти из функции `main()` или вызвать функцию `exit()`, необходимо установить условные переменные, чтобы они сигнализировали о возможном уничтожении объектов.[9]
- Завершать программу с помощью функции `quick_exit()`, предназначенной именно для того, чтобы завершать программу без вызова деструкторов глобальных и статических объектов (см. раздел 5.8.2).

Поскольку `std::cin`, `std::cout` и `std::cerr`, а также другие глобальные потоковые объекты (см. раздел 15.2.2) в соответствии со стандартом "не уничтожаются на протяжении выполнения программы", доступ к этим объектам в отсоединенных потоках не должен вызывать непредвиденных последствий. Однако могут возникнуть и другие проблемы, например промежуточные символы.

Тем не менее следует помнить, что единственным безопасным способом прекратить выполнение отсоединенного потока является вызов одной из функций "...`at_thread_exit()`", которые переводят главный поток в режим ожидания отсоединенного потока для истинного завершения программы. Если проигнорировать эту возможность, то, как написал один рецензент: "Отсоединенные потоки — это одна из тех тем, которые следует перенести в главу об опасных функциональных возможностях, в которых нет необходимости".

Идентификаторы потоков

Как видим, программа выводит идентификатор потока, полученный либо от объекта потока, либо внутри потока, используя пространство имен `this_thread` (определенное в заголовочном файле `<thread>`):

```
void doSomething (int num, char c)
{
   ...
   cerr << "THREAD-EXCEPTION (thread "
        << this_thread::get_id() << ")" << endl;
   ...
}

thread t(doSomething,5,'.');  // выводим пять точек в отдельном потоке
cout << "- started fg thread " << t1.get_id() << endl;
```

Этот идентификатор имеет специальный тип `std::thread::id`, который является уникальным для каждого потока. Кроме того, класс `id` имеет конструктор по умолчанию, который создает уникальный идентификатор для объекта, не являющегося потоком.

```
std::cout << "ID of \"no thread\": " << std::thread::id()
          << std::endl;
```

[9] В идеале следовало бы использовать функцию `notify_all_at_thread_exit()` (см. раздел 18.6.4), чтобы установить условную переменную, гарантирующую, что все локальные переменные в потоке будет уничтожены.

Для идентификаторов потоков допускаются только операции сравнения и потокового вывода. Не следует делать никаких дальнейших предположений, например, "объект, не являющийся потоком, имеет идентификатор, равный 0, а главный поток — 1". Реализация должна генерировать идентификаторы по запросу на лету, а не при запуске потоков, поэтому номер потока зависит от количество предыдущих запросов на идентификаторы потоков. Итак, код

```
std::thread t1(doSomething,5,'.');
std::thread t2(doSomething,5,'+');
std::thread t3(doSomething,5,'*');
std::cout << "t3 ID: " << t3.get_id() << std::endl;
std::cout << "main ID: " << std::this_thread::get_id() << std::endl;
std::cout << "nothread ID: " << std::thread::id() << std::endl;
```

может вывести

```
t3 ID: 1
main ID: 4
nothread ID: 0
```

или

```
t3 ID: 3
main ID: 4
nothread ID: 0
```

или

```
t3 ID: 1
main ID: 2
nothread ID: 3
```

или даже символы как идентификаторы потоков.

Следовательно, единственным способом идентифицировать поток, например главный поток, является сравнение его идентификатора с сохраненным идентификатором при запуске.

```
std::thread::id masterThreadID;
void doSomething()
{
   if (std::this_thread::get_id() == masterThreadID) {
       ...
   }
   ...
}

std::thread master(doSomething);
masterThreadID = master.get_id();
...
std::thread slave(doSomething);
...
```

Идентификаторы прекращенных потоков можно использовать снова.

Детально класс `thread` описан в разделе 18.3.6.

18.2.2. Обещания

Рассмотрим, как можно передавать параметры и обработку исключений между потоками (это позволит нам понять, как реализован высокоуровневый интерфейс, такой как `async()`). Конечно, для того чтобы передать значения в поток, можно просто передать их как аргументы. Если вам нужен результат, вы можете передать *возвращаемые аргументы* по ссылке, как описано в разделе 18.1.2).

Однако существует другой механизм для передачи результатов и исключений в качестве итога выполнения потока: класс `std::promise`. *Обещание* является аналогом фьючерса. Оба они могут временно иметь *общее состояние*, представляющее результат или исключение. В то время как фьючерс позволяет извлечь данные (с помощью функции `get()`), *обещание* позволяет ввести данные (используя одну из функций `set_...()`). Эту возможность демонстрирует следующий пример:

```
// concurrency/promise1.cpp

#include <thread>
#include <future>
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>
#include <functional>
#include <utility>

void doSomething (std::promise<std::string>& p)
{
   try {
      // вводим символ и генерируем исключение, если символ равен 'x'
      std::cout << "read char ('x' for exception): ";
      char c = std::cin.get();
      if (c == 'x') {
          throw std::runtime_error(std::string("char ")+c+" read");
      }
      ...
      std::string s = std::string("char ") + c + " processed";
      p.set_value(std::move(s)); // сохраняем результат
   }
   catch (...) {
      p.set_exception(std::current_exception()); // сохраняем исключение
   }
}

int main()
{
   try {
      // запускаем поток, используя обещание для хранения результата
      std::promise<std::string> p;
      std::thread t(doSomething,std::ref(p));
      t.detach();
      ...
```

```
    // создаем фьючерс для обработки результата
    std::future<std::string> f(p.get_future());

    // обрабатываем результат
    std::cout << "result: " << f.get() << std::endl;
  }
  catch (const std::exception& e) {
    std::cerr << "EXCEPTION: " << e.what() << std::endl;
  }
  catch (...) {
    std::cerr << "EXCEPTION " << std::endl;
  }
}
```

После включения заголовочного файла `<future>`, в котором объявлены обещания, можно объявить объект обещания, специализированный типом хранящегося или возвращаемого значения (или `void`, если таких значений нет).

```
std::promise<std::string> p; // хранит строку результата или исключение
```

Обещание создает *общее состояние* (см. раздел 18.3), которое можно использовать для хранения значения соответствующего типа или исключения. Его можно использовать во фьючерсе для извлечения этих данных в качестве результата работы потока.

Это обещание затем передается задаче, выполняемой как отдельный поток.

```
std::thread t(doSomething,std::ref(p));
```

Используя функцию `std::ref()` (см. раздел 5.4.3), мы гарантируем, что обещание передается по ссылке, так что существует возможность манипулировать его состоянием (копирование обещаний невозможно).

Теперь в потоке можно хранить значение или исключение, вызывая функцию `set_value()` или `set_exception()` соответственно.

```
void doSomething (std::promise<std::string>& p)
{
  try {
    ...
    p.set_value(std::move(s));                  // сохраняем результат
  }
  catch (...) {
    p.set_exception(std::current_exception()); // сохраняем исключение
  }
}
```

Для хранения исключения используется удобная функция `std::current_exception()`, определенная в заголовочном файле `<exception>` (см. раздел 4.3.3). Она возвращает исключение `std::exception_ptr`, обрабатываемое в текущий момент, или `nullptr`, если в настоящий момент никакое исключение не обрабатывается. Объект обещания хранит это исключение внутри себя.

В момент, когда мы сохраняем значение или исключение в общем *состоянии*, значение переходит в состояние *готовности*. Таким образом, это значение можно извлечь где-то еще. Однако для извлечения необходим соответствующий объект фьючерса, разделяющий

то же самое *общее состояние*. По этой причине, вызывая функцию `get_future()` из объекта обещания, мы создаем в функции `main()` объект фьючерса, имеющий обычную семантику, описанную в разделе 18.1. Мы могли бы также создать объект фьючерса до запуска потока.

```
std::future<std::string> f(p.get_future());
```

Теперь, используя функцию `get()`, мы сохраняем либо результат, либо исключение, которое генерируется заново (вызывая функцию `std::rethrow_exception()` для хранящегося исключения `exception_ptr`).

```
f.get(); // обрабатываем результат выполнения потока
```

Функция `get()` блокирует поток, пока *общее состояние* не перейдет в *состояние готовности*. Именно это происходит, когда из объекта обещания вызывается функция `set_value()` или `set_exception()`. Это *не* означает, что поток, установивший обещание, завершен. Поток может выполнять другие инструкции и даже сохранять дополнительные результаты в других объектах обещаний.

Если необходимо перевести *общее состояние* в *состояние готовности*, когда поток действительно завершен, — например, чтобы удалить локальные объекты и другие сущности в потоке до обработки результата, — вместо указанных выше функций следует вызвать функцию `set_value_at_thread_exit()` или `set_exception_at_thread_exit()`.

```
void doSomething (std::promise<std::string>& p)
{
  try {
      ...
      p.set_value_at_thread_exit(std::move(s));
   }
   catch (...) {
      p.set_exception_at_thread_exit(std::current_exception());
   }
}
```

Использование обещаний и фьючерсов не ограничено многопоточным режимом. Даже в однопоточных приложениях обещания можно использовать для хранения результата или исключения, которое мы хотим обработать позднее с помощью фьючерса.

Кроме того, в объекте обещания нельзя хранить значение и исключение одновременно. Любая попытка сделать это приведет к генерации исключения `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied` (см. раздел 4.3.1).

Дальнейшие подробности, касающиеся класса `promise`, изложены в разделе 18.3.4.

18.2.3. Класс packaged_task<>

Функция `async()` предоставляет программисту инструмент для работы с результатом выполнения задачи, которую он пытается немедленно запустить в фоновом режиме. Тем не менее иногда возникает необходимость обработать результат фоновой задачи, которую не обязательно запускать немедленно. Например, определять, когда и как следует осуществлять одновременное выполнение фоновых задач, может другой экземпляр, скажем, пул потоков. В этом случае вместо кода

```
double compute (int x, int y);

std::future<double> f = std::async(compute,7,5); // пытаемся запустить
                                                 // фоновую задачу
...
double res = f.get(); // ожидаем ее завершения и
                      // обрабатываем результат/исключение
```

можно написать

```
double compute (int x, int y);
std::packaged_task<double(int,int)> task(compute); // создаем задачу
std::future<double> f = task.get_future();         // получаем ее фьючерс
...
task(7,5);              // запускаем задачу (обычно в отдельном потоке)
...
double res = f.get();  // ожидаем ее завершения и обрабатываем результат/
исключение
```

в котором задача, как правило (хотя и не обязательно), запускается в отдельном потоке.

Таким образом, класс `std::packaged_task<>`, также определенный в заголовочном файле `<future>`, хранит как выполняемую сущность, так и ее возможный результат (так называемое *общее состояние* этой функциональной сущности; см. раздел 18.3).

Подробно класс `packaged_task` описан в разделе 18.3.5.

18.3. Подробное описание потоков

Итак, описав высоко- и низкоуровневый интерфейсы для (возможного) запуска потоков и работы с их результатами или исключениями, подведем итоги и перечислим еще не рассмотренные подробности.

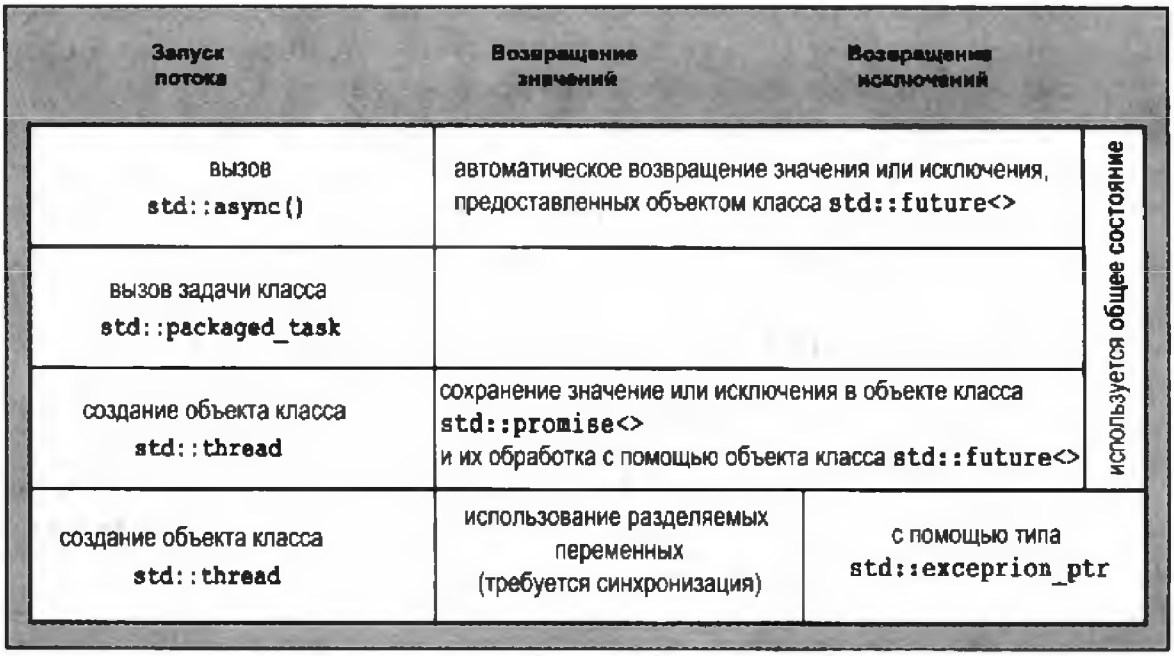


Рис. 18.1. Слои потоковых интерфейсов

С теоретической точки зрения интерфейсы состоят из слоев, предназначенных для запуска потоков и работы с их результатами или исключениями (рис. 18.1).

- Используя низкоуровневый интерфейс класса `thread`, можно запустить поток. Для того чтобы вернуть данные, необходимы разделяемые переменные (глобальные, статические или передаваемые как аргумент). Для того чтобы вернуть исключения, можно использовать объект типа `std::exception_ptr`, возвращаемый функцией `std::current_exception()`, и обработать функцией `std::rethrow_exception()` (см. раздел 4.3.3).
- Концепция *общего состояния* позволяет работать с возвращаемыми значениями или исключениями более удобным способом. С помощью низкоуровневого интерфейса класса `promise` можно создать *общее состояние*, которое можно обработать с помощью класса `future`.
- На более высоком уровне класс `packaged_task` или функция `async()` автоматически создает *общее состояние* и устанавливает оператор возврата или неперехваченное исключение.
- С помощью класса `packaged_task` можно создать объект с *общим состоянием*, для которого необходимо точно указать момент запуска потока.
- Функция `std::async()` позволяет не беспокоиться о конкретном моменте запуска потока. Известно лишь, что для получения результата необходимо вызвать функцию `get()`.

Общие состояния

Легко видеть, что основная концепция, используемая практически во всех этих функциональных механизмах, — *общее состояние*. Оно позволяет объектам, запускающим функциональные сущности в фоновом режиме и управляющим ими (обещанию упакованной задаче или функции `async()`) связываться с объектами, обрабатывающими их результат (фьючерсом или разделяемым фьючерсом). Таким образом, общее состояние позволяет хранить запускаемую функциональную сущность, информацию о состоянии и результат работы (возвращаемое значение или исключение).

Общее состояние переходит в *состояние готовности*, когда в нем хранится результат работы функциональной сущности (значение или исключение готовы к извлечению). Общее состояние обычно реализуется как объект с подсчетом ссылок, уничтожаемым в момент, когда из памяти *удаляется* последний ссылающийся на него объект.

18.3.1. Подробное описание функции `async()`

Как указано в разделе 18.1, функция `std::async()` представляет собой удобное средство для *возможного* запуска функциональной сущности в отдельном потоке. В результате появляется возможность выполнять функции в параллельном режиме, если базовая платформа это позволяет, в то же время не потеряв ничего, если платформа не поддерживает многопоточность.

Однако точное поведение функции `async()` является сложным и сильно зависит от стратегии запуска, которую можно передавать как первый аргумент. По этой причине каждую из трех версий функции `async()` можно вызвать так, как описано ниже с программистской точки зрения.

future **async** (`std::launch::async`, *F func*, *args*...)

- Пытается запустить функциональную сущность *func* со списком аргументов *args* как асинхронную задачу (параллельный поток).
- Если это невозможно, она генерирует исключение типа `std::system_error` с кодом ошибки `std::errc::resource_unavailable_try_again` (см. раздел 4.3.1).
- Если программа не прервет работу аварийно, то гарантируется, что запускаемый поток завершится до конца программы.
- Поток завершается в следующих случаях.
  - Если вызывается функция `get()` или `wait()` из возвращаемого фьючерса.
  - Если уничтожен последний объект, ссылающийся на *общее состояние*, представленное возвращаемым фьючерсом.
- Из этого следует, что вызов функции `async()` будет заблокирован, пока функция *func* не будет завершена и значение, возвращаемое функцией `async()`, не используется.

future **async** (`std::launch::deferred`, *F func*, *args*...)

- Передает функциональную сущность *func* со списком аргументов *args* как "отложенную" задачу, которая синхронно вызывается, когда вызывается функция `wait()` или `get()` из возвращаемого фьючерса.
- Если ни функция `wait()`, ни `get()` не вызывается, задача никогда не запускается.

future **async** (*F func*, *args*...)

- Является комбинацией вызова функции `async()` со стратегиями вызова `std::launch:async` и `std::launch::deferred`. В соответствии с текущей ситуацией выбирается одна из версий. Таким образом, функция `async()` *отложит* вызов функциональной сущности *func*, если немедленный вызов в стратегии запуска функции `async` невозможен.
- Таким образом, если функция `async()` может запустить новый поток для функциональной сущности *func*, она запускает *func*. В противном случае функциональная сущность *func* откладывается, пока из возвращаемого фьючерса не будет вызвана функция `get()` или `wait()`.
- Этот вызов гарантирует лишь, что после вызова функции `get()` или `wait()` из возвращаемого фьючерса функциональная сущность *func* будет вызвана и завершена.
- Без вызова функции `get()` или `wait()` из возвращаемого фьючерса функциональная сущность *func* может никогда не быть вызвана.
- Эта версия функции `async()` не будет генерировать исключение `system_error`, если она не может вызвать функциональную сущность *func* асинхронно (хотя по другим причинам она может генерировать системные ошибки).

Во всех этих версиях функции `async()` функциональная сущность *func* может быть *вызываемым объектом* (функцией, функцией-членом, функциональным объектом, лямбда-функцией; см. раздел 4.4). Примеры приведены в разделе 18.1.2.

Передача стратегии запуска `std::launch::async|std::launch::deferred` функции `async()` приводит к тем же последствиям, что и отсутствие стратегии. Передача `0` в качестве стратегии запуска приводит к непредвиденным последствиям (этот случай стандартной библиотекой C++ не предусмотрен, и разные реализации работают по-разному).

18.3.2. Подробное описание фьючерсов

Класс `future<>`[10], введенный в разделе 18.1, представляет *результат* операции. Он может быть возвращаемым значением или исключением, но не тем и другим одновременно. Результат направляется в *общее состояние*, которое обычно может создавать функция `std::async()`, `std::packaged_task` или обещание. В какой-то момент результат может еще не существовать; таким образом, фьючерс может хранить все необходимое для генерации результата.

Если фьючерс был возвращен функцией `async()` (см. раздел 18.3.1) и связанная с ней задача была *отложена*, то функции `get()` и `wait()` будут запущены синхронно. Отметим, что функции `wait_for()` и `wait_until()` не *запускают* отложенную задачу.

Результат можно извлечь только один раз. По этой причине фьючерс может находиться в корректном или некорректном состоянии: *корректное состояние* означает, что существует ассоциированная операция, для которой результат или исключение еще не извлечены.

В табл. 18.1 приведены операции над классом `future<>`.

Значение, возвращаемое функцией `get()`, зависит от типа `future<>`, специализированного следующими типами.

- Если это тип `void`, то функция `get()` также имеет тип `void` и ничего не возвращает.
- Если фьючерс параметризован ссылочным типом, то функция `get()` возвращает ссылку на возвращаемое значение.
- В противном случае функция `get()` возвращает копию или выполняет перемещающее присваивание возвращаемого значения в зависимости от того, поддерживает ли тип возвращаемого значения семантику перемещающего присваивания.

Функцию `get()` можно вызвать только один раз, поскольку она делает состояние фьючерса некорректным.

Если фьючерс, имеющий некорректное состояние, вызывает какую-нибудь функцию, кроме деструктора, оператора перемещающего присваивания или функции `valid()`, возникают непредсказуемые последствия. В этом случае стандарт рекомендует генерировать исключение типа `future_error` (см. раздел 4.3.1) с кодом ошибки `std::future_errc::no_state`, но это не требуется.

В классе нет ни копирующего конструктора, ни копирующей операции присваивания. Это гарантирует, что никакие два объекта не разделяют состояние с фоновой операцией. Состояние можно переместить в другой фьючерсный объект, только вызвав перемещающий конструктор или операцию перемещающего присваивания. Однако состояние фоновой задачи можно разделить между несколькими объектами с помощью объекта типа `shared_future`, создаваемого функцией `share()`.

Таблица 18.1. Операции класса `future<>`

Операция	Описание
future `f`	Конструктор по умолчанию; создает фьючерс с некорректным состоянием
future `f(`*rv*`)`	Перемещающий конструктор; создает новый фьючерс, получающий состояние аргумента *rv* и делающий некорректным состояние аргумента *rv*

[10] В ходе процесса стандартизации класс сначала назывался `unique_future`.

Окончание табл. 18.1

Операция	Описание
f.~*future*()	Разрушает состояние и объект `*this`
f = *rv*	Перемещающее присваивание; разрушает старое состояние объекта *f*, присваивает состояние объекта *rv* и делает его некорректным
f.`valid()`	Возвращает `true`, если объект находится в корректном состоянии, так что можно вызвать следующие функции-члены
f.`get()`	Блокирует поток, пока выполняется фоновая операция (инициируя синхронный запуск отложенной ассоциированной функциональной сущности), возвращает результат (если он существует) или генерирует любое возникающее исключение и делает состояние некорректным
f.`wait()`	Блокирует поток, пока выполняется фоновая операция (вынуждая синхронный запуск отложенной ассоциированной функциональной сущности)
f.`wait_for(`*dur*`)`	Блокирует поток на период, заданный аргументом *dur*, или на период выполнения фоновой операции (запуск отложенного потока не вынуждается)
f.`wait_until(`*tp*`)`	Блокирует поток до момента *tp* или на период выполнения фоновой операции (запуск отложенного потока не вынуждается)
f.`share()`	Возвращает объект класса `shared_future` с текущим состоянием и делает некорректным состояние объекта *f*

Если вызывается деструктор фьючерса, который был последним владельцем общего состояния, и ассоциированная задача была запущена, но еще не завершена, то деструктор блокирует поток до конца выполнения задачи.

18.3.3. Подробное описание разделяемых фьючерсов

Класс `shared_future<>` (введенный в разделе 18.1.3) реализует ту же семантику и интерфейсы, что и класс `future` (см. раздел 18.3.2), за исключением следующих отличий.

- Разрешено несколько вызовов функции `get()`. Таким образом, функция `get()` не делает состояние фьючерса некорректным.
- Поддерживается семантика копирования (копирующий конструктор, копирующая операция присваивания).
- `get()` — *константная* функция-член, возвращающая константную ссылку на значение, хранящееся в *общем состоянии* (т.е. программист должен гарантировать, что продолжительность существования возвращаемой ссылки короче времени существования *общего состояния*). Если класс `std::future` не специализирован ссылочным типом, функция-член `get()` является *неконстантной* функцией-членом, возвращающей копию, которая присваивается путем перемещения (или просто копируется, если перемещение не поддерживается).
- Функция-член `share()` не предусмотрена.

Тот факт, что значение, возвращаемое функцией `get()`, не копируется, создает определенные риски. Помимо аспектов, касающихся продолжительности существования объектов, возможно состязание за данные. Это явление возникает в ситуациях, когда порядок

выполнения конфликтующих действий над одними и теми же данными не определен. Например, это относится к несинхронизированному вводу и выводу в многопоточных средах. В результате возникают непредсказуемые последствия (см. раздел 18.4.1).

Исключения создают такие же проблемы. В процессе стандартизации обсуждался пример, в котором исключение перехватывалось по ссылке, а затем модифицировалось.

```
try {
    shared_future<void> sp = async(f);
    sp.get();
}
catch (E& e) {
    e.modify(); // риск непредсказуемых последствий из-за состязания за данные
}
```

Этот код создает состязание за данные, если исключение обрабатывается другим потоком. Для решения этой проблемы было предложено потребовать, чтобы функции `current_exception()` и `rethrow_exception()`, которые используются для передачи исключений между потоками, создавали копии исключений. Однако стоимость этого изменения была сочтена слишком высокой. В результате ответственность за последствия была возложена на программистов: они сами должны знать, что делают, работая с неконстантными ссылками, используемыми в разных потоках.

18.3.4. Подробное описание класса `std::promise`

Объект класса `std::promise`, введенный в разделе 18.2.2, предназначен для временного хранения (возвращаемого) значения или исключения. В принципе обещание может хранить *общее состояние* (см. раздел 18.3). Если *общее состояние* хранит значение или исключение, говорят, что оно переходит в *состояние готовности*.

Операции, доступные для класса `promise`, приведены в табл. 18.2

Таблица 18.2. Операции над объектами класса `promise`

Операция	Описание
promise *p*	Конструктор по умолчанию. Создает обещание с общим состоянием
promise *p*`(allocator_arg,` *alloc*`)`	Создает обещание с общим состоянием, которое использует в качестве механизма распределения памяти аргумент *alloc*
promise *p* (*rv*)	Перемещающий конструктор. Создает новый объект обещания, получающий новое состояние аргумента *rv* и удаляющий *общее состояние* из аргумента *rv*
p.~***promise***`()`	Освобождает *общее состояние* и, если оно не находилось в состоянии готовности (т.е. не содержало ни значения, ни исключения), сохраняет исключение `std::future_error` с условием `broken_promise`
p = *rv*	Перемещающее присваивание. Присваивает и перемещает состояние объекта *rv* в объект *p* и, если объект *p* не был в состоянии готовности, сохраняет исключение `std::future_error` с условием `broken_promise`

Окончание табл. 18.2

Операция	Описание
`swap(`*p1*`,`*p2*`)`	Обменивает состояния *p1* и *p2*
p1`.swap(`*p2*`)`	Обменивает состояния *p1* и *p2*
p`.get_future()`	Создает фьючерсный объект для извлечения *общего состояния* (результата работы потока)
p`.set_value(`*val*`)`	Устанавливает аргумент *val* как (возвращаемое) значение и переводит обещание в состояние *готовности* (или генерирует исключение `std::future_error`)
p`.set_value_at_thread_exit(`*val*`)`	Устанавливает аргумент *val* как (возвращаемое) значение и переводит обещание в состояние *готовности* по завершении потока (или генерирует исключение `std::future_error`)
p`.set_exception(`*e*`)`	Устанавливает объект *e* в качестве исключения и переводит обещание в состояние *готовности* (или генерирует исключение `std::future_error`)
p`.set_exception_at_thread_exit(`*e*`)`	Устанавливает объект *e* в качестве исключения и переводит обещание в состояние *готовности* по завершении потока (или генерирует исключение `std::future_error`)

Функцию `get_future()` можно вызвать только один раз. Второй вызов приведет к генерации исключения `std::future_error` с кодом ошибки `std::future_errc::future_already_retrieved`. В принципе, если с обещанием не связано никакое *общее состояние*, может быть сгенерировано исключение `std::future_error` с кодом ошибки `std::future_errc::no_state`.

Все функции-члены, устанавливающие значение или исключение, являются безопасными для потоков. Иначе говоря, они ведут себя так, будто мьютексы гарантируют, что в каждый момент времени только одна из них может изменить *общее состояние*.

18.3.5. Подробное описание класса `std::packaged_task`

Класс `std::packaged_task<>` предназначен для хранения как функциональной сущности, подлежащей выполнению, так и результата ее работы (так называемого *общего состояния* функциональной сущности, описанного в разделе 18.3), которым может быть возвращаемое значение или исключение, сгенерированное этой функциональной сущностью. Программист может инициализировать упакованную задачу с ассоциированной функциональной сущностью. В этом случае можно вызвать эту функциональную возможность, вызвав операцию `()` для упакованной задачи. В заключение можно обработать результат, получив фьючерс для упакованной задачи. Список операций, предусмотренных для класса `packaged_task`, приведен в табл. 18.3.

Любое исключение, вызванное конструктором, получающим задачу, например из-за нехватки памяти, также сохраняется в *общем состоянии*.

Попытка вызвать задачу или функцию `get_future()` в ситуации, когда ни одно состояние не является доступным, приводит к генерации исключения `std::future_error` (см. раздел 4.3.1) с кодом ошибки `std::future_errc::no_state`. Второй вызов

функции `get_future()` приводит к генерации исключения типа `std::future_error` с кодом ошибки `std::future_errc::future_already_retrieved`. Второй вызов задачи приводит к генерации исключения `std::future_error` с кодом ошибки `std::future_errc::promise_already_satisfied`.

Таблица 18.3. Операции класса `packaged_task<>`

Операция	Описание
packaged_task *pt*	Конструктор по умолчанию. Создает упакованную задачу без общего состояния и хранящейся в нем задачи
packaged_task *pt*(*f*)	Создает объект для задачи *f*
packaged_task *pt*(*alloc*,*f*)	Создает объект для задачи *f* с помощью распределителя памяти *alloc*
packaged_task *pt*(*rv*)	Перемещающий конструктор. Перемещает упакованную задачу *rv* (задачу и состояние) в объект *pt* (объект *rv* после этого уже не будет иметь *общего состояния*)
pt.~***packaged_task***()	Уничтожает объект `*this` (может перевести *общее состояние* в состояние готовности)
pt = *rv*	Перемещающее присваивание. Выполняет перемещающее присваивание и перемещает упакованную задачи *rv* (задачу и состояние) объекту *pt* (объект *rv* после этого уже не будет иметь *общего состояния*)
`swap`(*pt1*,*pt2*)	Обменивает упакованные задачи
pt1.`swap`(*pt2*)	Обменивает упакованные задачи
pt.`valid()`	Возвращает `true`, если объект *pt* имеет *общее состояние*
pt.`get_future()`	Возвращает фьючерс для извлечения *общего состояния* (результата выполнения задачи)
pt(*args*)	Вызывает задачу (с необязательными аргументами) и переводит *общее состояние* в состояние готовности
pt.`make_ready_at_thread_exit`(*args*)	Вызывает задачу (с необязательными аргументами) и после выхода из потока переводит *общее состояние* в состояние готовности
pt.`reset()`	Создает новое *общее состояние* для объекта *pt* (может перевести старое *общее состояние* в состояние готовности)

Деструктор и функция `reset()` *аннулируют общее состояние*, иначе говоря, упакованная задача удаляет *общее состояние* из памяти и, если *общее состояние* еще не перешло в состояние готовности, переводит его в это состояние, сохраняя в качестве результата исключение `std::future_error` с кодом ошибки `std::future_errc::broken_promise`.

Как обычно, функция `make_ready_at_thread_exit()` позволяет гарантировать удаление локальных объектов и других сущностей, когда поток завершает задачу до обработки результатов.

18.3.6. Подробное описание класса `std::thread`

Объект класса `std::thread`, введенного в разделе 18.2.1, предназначен для запуска и представления потока. Эти объекты должны однозначно соответствовать потокам, предоставляемым операционной системой. Список операций для класса `thread` приведен в табл. 18.4.

Таблица 18.4. Операции над объектами класса `thread`

Операция	Описание
thread t	Конструктор по умолчанию. Создает *неприсоединенный* объект потока
thread t (*f*, ...)	Создает объект потока, представляющий аргумент *f*, запускаемый как поток (с дополнительными аргументами), или генерирует исключение `std::system_error`
thread t (*rv*)	Перемещающий конструктор. Создает новый объект потока, получающий состояние аргумента *rv*, и делает поток *rv* *неприсоединяемым*
t.~***thread***()	Уничтожает объект `*this`. Вызывает функцию `std::terminate()`, если объект является *присоединяемым*
t = *rv*	Перемещающее присваивание. Присваивает и перемещает состояние потока *rv* потоку *t* или вызывает функцию `std::terminate()`, если объект *t* является *присоединяемым*
t.`joinable()`	Возвращает значение `true`, если объект *t* имеет ассоциированный поток (т.е. является *присоединяемым*)
t.`join()`	Ожидает завершения ассоциированного потока (если поток не является *присоединяемым*, генерирует исключение `std::system_error`) и делает этот объект *неприсоединяемым*
t.`detach()`	Разрывает связь между объектом *t* и его потоком, пока поток выполняет работу (если поток не является *присоединяемым*, генерирует исключение `std::system_error`), и делает объект *неприсоединяемым*
t.`get_id()`	Возвращает уникальный идентификатор типа `std::thread::id`, если поток является *присоединяемым*, и `std::thread::id()` — в противном случае.
t.`native_handle()`	Возвращает объект типа `native_handle_type`, зависящий от реализации, для непереносимых исключений
`t1.swap(t2)`	Обменивает состояния потоков `t1` и `t2`
`swap(t1,t2)`	Обменивает состояния потоков `t1` и `t2`

Связь между объектом потока и самим потоком возникает в момент инициализации (или при выполнении перемещающего копирования/присваивания) *вызываемого объекта* (см. раздел 4.4) необязательными дополнительными аргументами. Связь разрывается либо функцией `join()` (ожидающей результата выполнения потока), либо функцией

detach() (явно разрывающей связь с потоком). И та и другая функция должна быть вызвана до завершения существования объекта потока или до перемещающего присваивания нового потока. В противном случае программа аварийно завершит выполнение с помощью функции std::terminate() (см. раздел 5.8.2).

Если объект потока связан с потоком, то говорят, что он является *присоединяемым* (joinable). В этом случае функция joinable() возвращает true, а функция get_id() возвращает идентификатор потока, отличающийся от std::thread::id().

Идентификаторы потоков имеют собственный тип std::thread::id. Его конструктор по умолчанию возвращает уникальный идентификатор, означающий, что потока нет. Функция thread::get_id() возвращает это значение, если с объектом не ассоциирован ни один поток или если потоку присвоен другой идентификатор (т.е. он является *присоединяемым*). Для идентификаторов потока предусмотрены только операции сравнения и записи в поток вывода. Кроме того, предусмотрена функция хеширования для управления идентификаторами потока в неупорядоченных контейнерах (см. раздел 7.9). Идентификатор прекращенного потока можно использовать снова. Не следует делать никаких других предположений об идентификаторах потоков, особенно об их значениях (см. раздел 18.2.1).

Отсоединенные потоки не должны иметь доступа к объектам, срок существования которых истек. По этой причине по завершении программы необходимо гарантировать, что отсоединенные потоки не имеют доступа к глобальным или статическим объектам (см. раздел 18.2.1).

Кроме того, класс std::thread имеет статическую функцию-член для запроса подсказки о возможном количестве параллельных потоков.

unsigned int **std::thread::hardware_concurrency** ()

- Возвращает количество возможных потоков.
- Это число является всего лишь подсказкой и не обязательно должно быть точным.
- Если число не определяется или определено приблизительно, возвращает 0.

18.3.7. Пространство имен this_thread

Для любого потока, включая главный, в заголовочном файле <thread> объявлено пространство имен std::this_thread, в котором предусмотрены потоковые глобальные функции (табл. 18.5).

Таблица 18.5. Потоковые операции в пространстве имен std::this_thread std::this_thread

Операция	Описание
this_thread::get_id()	Возвращает идентификатор текущего потока
this_thread::sleep_for(*dur*)	Блокирует поток на период *dur*
this_thread::sleep_until(*tp*)	Блокирует поток до момента *tp*
this_thread::yield()	Подсказывает изменение момента запуска следующего потока

Функции sleep_for() и sleep_until() обычно отличаются способами работы с корректировками системного времени (см. раздел 5.7.5).

Операция `this_thread::yield()` подсказывает системе, что было бы полезно урезать оставшееся время выполнения потока, чтобы среда выполнения программ могла изменить расписание и запустить другие потоки. Как правило, это нужно для того, чтобы подождать или опросить другой поток (см. раздел 18.1.1) или установить атомарный флаг с помощью другого потока (см. раздел 18.4.3)[11].

```
while (!readyFlag) { // цикл до готовности данных
   std::this_thread::yield();
}
```

Аналогичная ситуация возникает, когда не удается получить блокировку или мьютекс при одновременной установке нескольких блокировок или мьютексов. В этом случае можно ускорить выполнение приложения, выполнив функцию `yield()` до попытки установить блокировки и мьютексы в другом порядке[12].

18.4. Синхронизация потоков, или проблема конкурентности

Использование нескольких потоков практически всегда связано с конкурентным доступом к данным. Несколько потоков редко работают независимо друг от друга. Потоки могут передавать данные другим потокам или запускать другие процессы.

Именно этот аспект обусловливает сложность многопоточных режимов. Многое может стать неправильным. Иначе говоря, многое может оказаться неожиданным для наивных (и даже опытных) программистов.

Итак, прежде чем рассмотреть разные способы синхронизации потоков и конкурентный доступ к данным, необходимо понять существо проблемы. Затем мы обсудим следующие приемы синхронизации потоков.

- Мьютексы и блокировки (см. раздел 18.5), включая `call_once()` (см. раздел 18.5.3).
- Условные переменные (см. раздел 18.6).
- Атомарные операции (см. раздел 18.7).

18.4.1. Осторожно, конкурентность!

Прежде чем рассмотреть подробности проблемы конкурентности, сформулируем первое правило, которому необходимо следовать, если вы хотите начинать программировать, не углубляясь в этот раздел. Правило работы с несколькими потоками можно сформулировать следующим образом.

Единственный безопасный способ обеспечить конкурентный доступ в одним и тем же данным от нескольких потоков без синхронизации — ***ВСЕ*** потоки должны только ***ЧИТАТЬ*** данные.

Под "одними и теми же данными" здесь подразумевается использование одного и того же участка памяти. Если разные потоки одновременно используют *разные* переменные

[11] Благодарю Бартоша Милевски (Bartosz Milewski) за этот пример.

[12] Благодарю Ховарда Хиннанта (Howard Hinnant) за этот пример.

или объекты или разные их члены, то никаких проблем нет, потому что по стандарту C++11 каждая переменная, за исключением битового поля, гарантированно имеет свой собственный участок памяти[13]. Единственным исключением является битовое поле, потому что разные битовые поля могут использовать одну и ту же область памяти, а значит, разные битовые поля имеют общий доступ к одним и тем же данным.

Однако, если несколько потоков имеют конкурентный доступ к *одной и той же* переменной, или объекту, или члену объекта и хотя бы один из потоков выполняет модификации, можно легко создать крупные неприятности, если не синхронизировать их доступ. В языке C++ эта ситуация называется *состязанием за данные*. В стандарте C++11 *состязание за данные* определено как "два конфликтующих действия в разных потоках, причем хотя бы одно из них не является атомарным, и до следующего действия ничего не происходит". Состязание за данные всегда приводит к непредсказуемым последствиям.

Во всех ситуациях, связанных с состязанием за потоки, проблема заключается в том, что программа может часто делать то, что от нее ожидают, но это происходит *не всегда*. Это одна из самых ужасных проблем программирования. Использование других данных, переход в продуктивный режим или изменение платформы может внезапно разрушить работоспособность программы. Следовательно, при работе с несколькими потоками целесообразно побеспокоиться о состязании за данные.

18.4.2. Причина проблем при состязании за данные

Для того чтобы понять проблемы, которые порождаются состязанием за данные, необходимо разобраться с тем, что гарантирует язык C++ для параллельной работы. Такой язык программирования, как C++, всегда абстрактно поддерживает разные платформы и аппаратное обеспечение, предоставляющие разные возможности и интерфейсы в соответствии с их архитектурой и предназначением. Таким образом, стандарт такого языка, как C++, регламентирует *действие* инструкций и операций, а не соответствующего сгенерированного ассемблерного кода. Стандарт отвечает на вопрос "*что*", а не "*как*".

В принципе поведение не определяется настолько точно, чтобы не возникало разных возможностей для его реализации. На самом деле поведение может быть преднамеренно оставлено неопределенным. Например, порядок вычисления аргументов при вызове функции не определен. Программа, ожидающая конкретного порядка вычислений, может вызвать непредвиденные последствия.

Итак, возникает важный вопрос: какие гарантии дает язык? Программисты не должны ожидать большего, даже несмотря на то, что дополнительные гарантии могут выглядеть очевидными. Фактически в соответствии с *правилом* "*как будто*" каждый компилятор может оптимизировать код, хотя внешне работа программы будет выглядеть по-прежнему. Следовательно, сгенерированный код — это "*черный ящик*", содержимое которого может изменяться в то время, как его *наблюдаемое поведение* остается неизменным. Процитируем стандарт C++.

"Реализация может свободно игнорировать любое требование данного международного стандарта, если результирующее поведение программы выглядит так, "*как будто*" это требование было выполнено. Например, фактическая реализация не обязана вычислять

[13] Гарантированное использование разных участков памяти разными объектами до принятия стандарта C++11 не поддерживалось. Стандарт C++98/C++03 рассматривал только однопоточные приложения. Таким образом, строго говоря, до принятия стандарта C++11 конкурентный доступ к нескольким объектам приводил к непредсказуемым последствиям, но на практике обычно не вызывал проблем.

часть выражения, которая в дальнейшем не используется, и при этом не возникают побочные эффекты".

Любое неопределенное поведение дает разработчикам компиляторов и аппаратного обеспечения свободу и возможность генерировать оптимизированный код, следуя критерию качества. Это относится как к компиляторам, так и к аппаратному обеспечению: компиляторы могут разворачивать циклы, переупорядочивать операторы, исключать "мертвый код", предварительно выбирать данные, а современное аппаратное обеспечение, например буфера, могут переупорядочивать нагрузку и память.

Переупорядочение может как повысить скорость выполнения программы, так и нарушить ее работу. Для того чтобы извлечь пользу из повышения быстродействия, часто жертвуют безопасностью. Таким образом, необходимо понимать, что гарантирует стандарт, особенно в ситуации состязания за данные.

18.4.3. Что именно создает опасность (расширение проблемы)

Для того чтобы компиляторы и аппаратное обеспечение могли свободно оптимизировать код, язык C++ *не предусматривает* ожидаемых гарантий. Причина заключается в том, что распространение этих гарантий на все возможные ситуации могло бы слишком сильно снизить производительность программ. В программах на языке C++ могут возникнуть следующие проблемы.

- **Несинхронизированный доступ к данным**. Когда два потока параллельно читают и записывают одни и те же данные, остается неясным, какой оператор должен выполняться первым.
- **Наполовину записанные данные**. Когда один поток читает данные, а другой их модифицирует, читающий поток может даже прочитать данные *посередине* процесса записи, выполняемого другим потоком. При этом поток не считывает ни старые, ни новые значения.
- **Переупорядоченные операторы**. Операторы и операции могут быть переупорядочены, так что поведение каждого потока в отдельности остается корректным, но сочетание *всех* потоков нарушает работу программы в целом.

Несинхронизированный поток данных

Следующий код гарантирует, что функция `f()` вызывается для абсолютного значения аргумента `val` и меняет его знак, если число является отрицательным:

```
if (val >= 0) {
   f(val);   // передает положительное значение val
}
else {
   f(-val);  // передает значение val с противоположным знаком
}
```

В однопоточной среде этот код работает прекрасно, а в многопоточной работает не всегда. Если доступ к значению `val` имеют несколько потоков, значение `val` может измениться между выполнением оператора `if` и вызовом функции `f()`, поэтому функции `f()` будет передано значение `val` с противоположным знаком.

По той же причине следующий код может вызвать проблемы:

```
std::vector<int> v;
...
if (!v.empty()) {
   std::cout << v.front() << std::endl;
}
```

Если объект `v` используется несколькими потоками, то между вызовами функций `empty()` и `front()` он может стать пустым и вызвать непредвиденные последствия (см. раздел 7.3.2).

Эта проблема также относится к коду, реализующему функцию из стандартной библиотеки C++. Например, гарантия, что код

```
v.at(5); // возвращает значение элемента с индексом 5
```

генерирует исключение, если объект `v` не содержит достаточное количество элементов, больше не распространяется на ситуацию, в которой поток изменяет объект `v` во время вызова функции `at()`. Таким образом, следует иметь в виду следующее.

> **Если не указано противоположное, то функции из стандартной библиотеки C++ обычно не поддерживают конкурентные операции чтения и записи одной и той же структуры данных[14].**

Другими словами, если не указано обратное, многократные вызовы одного и того же объекта из разных потоков приводят к непредсказуемым последствиям.

Однако стандартная библиотека C++ предоставляет некоторые гарантии, касающиеся безопасности потоков (см. раздел 4.5).

- Возможен параллельный доступ к *разным элементам* одного и того же контейнера (за исключением класса `vector<bool>`). Таким образом, разные потоки могут параллельно читать и/или записывать разные элементы одного и того же контейнера. Например, каждый поток может делать нечто и хранить результат в своем элементе общего вектора.
- Параллельный доступ к строковому, файловому потоку или буферу приводит к непредсказуемым последствиям.

Однако, как было показано выше, возможен форматированный ввод и вывод в стандартном потоке, синхронизированный с механизмом ввода-вывода языка C (см. раздел 15.14.1), хотя при этом могут возникнуть промежуточные символы.

Наполовину записанные данные

Допустим, у нас есть переменная[15]

```
long long x = 0;
```

[14] Как указал Ганс Боэм (Hans Boehm), поддержка конкурентного доступа к библиотечным объектам в принципе может быть вредной, потому что необходимость синхронизировать доступ к структурам данных не сводится к управлению отдельными механизмами доступа и требует защиты более крупных фрагментов программы. Это означает, что программисты должны реализовывать свои собственные блокировки и библиотечные блокировки могут оказаться излишними.

[15] Этот пример с разрешения взят из работы [*N2480:MemMod*].

один поток, записывающий данные,

```
x = -1;
```

и один поток, считывающий данные,

```
std::cout << x;
```

Что выведет программа? Иначе говоря, какое значение прочитает второй поток, если он выводит переменную x? Возможны следующие ответы:

- 0 (старое значение переменной x), если первый поток еще не присвоил ей значение -1;
- -1 (новое значение переменной x), если первый поток уже присвоил ей значение -1;
- *любое другое значение*, если второй поток читает x во время присваивания -1 первым потоком.

Последний вариант — *любое другое значение* — вполне возможен, например, если на 32-битовом компьютере результаты 64-битового присваивания хранятся в двух 32-битовых блоках памяти и чтение вторым потоком происходит в момент, когда первый блок уже записан, а второй еще нет.

Учтите, что это относится не только к типу `long long`. Даже для элементарных типов данных, таких как `int` и `bool`, стандарт *не гарантирует*, что чтение или запись являются *атомарными*; иначе говоря, чтение и запись имеют эксклюзивный доступ к данным, не допускающий прерываний. Вероятность состязания за данные можно снизить, но, для того чтобы совсем исключить его, требуется предпринять ряд защитных мер.

То же самое можно сказать о более сложных структурах данных, даже если они предоставлены стандартной библиотекой C++. Например, при работе с классом `std::list<>` (см. раздел 7.5) именно программист должен гарантировать, что список не будет модифицирован другим потоком, когда поток вставляет или удаляет элементы. В противном случае поток может перевести список в некорректное состояние, в котором, например, указатель на следующий элемент уже изменен, а указатель на предыдущий — нет.

Переупорядоченные инструкции

Рассмотрим другой простой пример[16]. Допустим, у нас есть два общих объекта: переменная типа `int` для передачи данных от одного потока другому и булева переменная `readyFlag`, сигнализирующая о том, что первый поток приготовил данные.

```
long data;
bool readyFlag = false;
```

Наивный подход к решению этой задачи заключается в синхронизации присвоения переменной `data` в одном потоке с ее получением в другом потоке. Таким образом, поток-отправитель выполняет инструкции

```
data = 42;
readyFlag = true;
```

[16] Этот пример взят из многочисленных статей, опубликованных на сайте *Bartosz Milewski's Programming Cafe* (см. [*Milewski:Multicore*] и [*Milewski:Atomics*]).

а поток-получатель выполняет инструкции

```
while (!readyFlag) { // цикл, пока данные не будут готовы
   ;
}
foo(data);
```

Не зная никаких деталей, почти каждый программист сначала предположит, что второй поток вызывает функцию `foo()`, когда переменная `data` имеет значение `42`, считая, что вызов функции `foo()` можно выполнить только в случае, когда переменная `readyFlag` равна `true`, что в свою очередь может произойти только после того, как первый поток присвоит переменной `data` значение `42`, потому что присваивание происходит раньше.

Однако это не так. Фактически второй поток может вывести значение `data` *до* того, как первый поток присвоит значение `42` (и даже любое другое число, потому что присвоение значения `42` может произойти не полностью).

Другими словами, компилятор и/или аппаратное обеспечение могут переупорядочить инструкции, так что на самом деле они будут выполняться в следующем порядке:

```
readyFlag = true;
data = 42;
```

В целом такие правила языка C++ допускают такое переупорядочение, требуя лишь, чтобы *наблюдаемое поведение внутри потока сгенерированного кода* было корректным. Поведение первого потока не зависит от того, какая переменная будет модифицирована первой: `readyFlag` или `data`; с точки зрения этого потока они не зависят друг от друга. Таким образом, переупорядочение инструкций допускается, потому что внешне отдельный поток выглядит тем же самым.

По той же причине даже второй поток может переупорядочить инструкции, если поведение потока от них не зависит.

```
foo(data);
while (!readyFlag) { // цикл, пока данные не будут готовы
   ;
}
```

Отметим, что наблюдаемое поведение может зависеть от переупорядочения вызовов `foo()`, если оно генерирует исключения. Таким образом, оно зависит от того, разрешены ли такие переупорядочения или нет, т.е. проблема существует.

Причина, по которой такие модификации разрешены, заключается в том, что по умолчанию компиляторы языка C++ способны генерировать высокооптимизированный код, а некоторые способы оптимизации могут приводить к переупорядочению инструкций. По умолчанию процесс оптимизации не обязан следить за возможно существующими другими потоками. Это упрощает оптимизацию, ограничивая ее локальным анализом.

18.4.4. Способы решения проблем

Для решения трех основных проблем конкурентного доступа к данным используются следующие концепции.

- **Атомарность**. Это значит, что чтение и запись переменной или последовательности инструкций происходит эксклюзивно и без прерываний, так что один поток не может вмешиваться в процесс изменения состояния другим потоком.

- **Порядок**. Нужны гарантии, что порядок конкретных инструкций или групп инструкций не будет изменен.

Стандартная библиотека C++ предоставляет самые разные средства для работы с этими концепциями, так что программы могут получить преимущества от дополнительных гарантий, касающихся конкурентного доступа.

- Можно использовать *фьючерсы* (см. раздел 18.1) и *обещания* (см. раздел 18.2.2), гарантирующие атомарность и порядок. Задание *результата* (возвращаемого значения или исключения) *общего состояния* гарантированно происходит до его обработки. Следовательно, доступ для чтения и записи не открывается одновременно.
- Можно использовать *мьютексы* и *блокировки* (см. раздел 18.5) для работы с *критичными разделами*, или *защищенными зонами*, к которым можно обеспечить эксклюзивный доступ, так что, например, между проверкой и операцией, основанной на этой проверке, ничего не может произойти. Блокировки обеспечивают атомарность за счет прекращения любого доступа к данным с помощью второй блокировки, пока первая блокировка, установленная на тот же самый ресурс, не будет снята. Точнее говоря, снятие блокировки с объекта, принадлежащего одному из потоков, гарантированно происходит до того, как этим объектом завладеет другой поток. Однако, если оба потока используют блокируемый доступ к данным, порядок их доступа от запуска к запуску может изменяться.
- Можно использовать *условные переменные* (см. раздел 18.6), позволяющие одному потоку ожидать, пока предикат, управляемый другим потоком, не станет истинным. Это происходит при управлении порядком выполнения нескольких потоков, позволяя одному или нескольким потокам обрабатывать данные или состояние, созданные другими потоками[17].
- Можно использовать *атомарные типы данных* (см. раздел 18.7), для того чтобы гарантировать, что каждое обращение к переменной или объекту является атомарным, а порядок операций над атомарными типами остается неизменным.
- Можно использовать *низкоуровневый интерфейс атомарных типов данных* (см. раздел 18.7.4), позволяющий экспертам ослабить порядок выполнения атомарных операций или использовать минимальные *барьеры* для доступа к памяти (fences).

Как видите, этот список организован по принципу "сверху вниз". Высокоуровневые средства, такие как фьючерсы и обещания, а также мьютексы и блокировки, просты в использовании и безопасны. Низкоуровневые средства, такие как атомарность и особенно низкоуровневый интерфейс, могут обеспечить более высокую производительность, потому что они имеют меньшее время задержки и, следовательно, являются более универсальными. Однако риск их неправильного использования значительно возрастает. Тем не менее низкоуровневые средства обеспечивают простые решения конкретных высокоуровневых проблем.

Используя атомарность, мы идем в направлении *программирования без блокировок*, которое даже эксперты часто понимают неправильно. Процитируем Герба Саттера (Herb Sutter) [*Sutter:LockFree*]: "[Код без блокировок] сложен даже для экспертов. Легко написать программу без блокировок, которая кажется работоспособной, но трудно написать

[17] Эксперты по параллельному программированию не желают признавать условные переменные инструментом решения проблем, возникающих при конкурентном доступе к данным, потому что они скорее предназначены для повышения эффективности, а не для защиты корректности работы потоков.

программу без блокировок, которая была бы правильной и хорошо работала. Даже хорошие и реферируемые журналы публиковали массу программ без блокировок, которые на самом деле содержали скрытые ошибки и требовали исправления".

Ключевое слово volatile и параллельность

Обратите внимание на то, что я нигде не употреблял слово `volatile` в контексте конкурентного доступа к данным, хотя, возможно, вы этого ждали. Это объясняется следующими причинами.

- В языке C++ ключевое слово `volatile` используется для предотвращения слишком большого объема оптимизации.
- В языке Java ключевое слово `volatile` обеспечивает некоторые гарантии атомарности и порядка.

В языке C++ ключевое слово `volatile` "только" указывает на то, что доступ к внешним ресурсам, таким как разделяемая память, не следует оптимизировать. Например, без ключевого слова `volatile` компилятор мог бы исключить лишние загрузки в разделяемый сегмент памяти, потому что с ним не происходит никаких изменений в программе. Однако в языке C++ слово `volatile` не гарантирует ни атомарности, ни конкретного порядка[18]. Таким образом, семантика ключевого слова `volatile` в языках C++ и Java в настоящее время разная.

Объяснения, почему ключевое слово `volatile` обычно не требуется, когда для чтения данных в цикле используются мьютексы, приведены в разделе 18.5.1.

18.5. Мьютексы и блокировки

Мьютекс, или *взаимоисключающая блокировка*, — это объект, помогающий управлять конкурентным доступом к ресурсу, обеспечивая эксклюзивный доступ к нему. Ресурсом может быть объект или комбинация нескольких объектов. Для того чтобы получить эксклюзивный доступ к ресурсу, соответствующий поток *блокирует* мьютекс, предотвращая его блокировку со стороны других потоков, пока первый поток не разблокирует *мьютекс*.

18.5.1. Использование мьютексов и блокировок

Допустим, мы хотим защитить конкурентный доступ к объекту `val`, который используется в нескольких местах:

```
int val;
```

Наивный подход к синхронизации доступа заключается в использовании мьютекса для обеспечения доступа и управления им:

```
int val;
std::mutex valMutex; // управление эксклюзивным доступом к переменной val
```

[18] Благодарю Скотта Мейерса (Scott Meyers) за это замечание.

Затем каждое обращение должно блокировать этот мьютекс, чтобы получить эксклюзивный доступ. Например, один поток можно запрограммировать следующим образом (это плохое решение, которое мы будем улучшать):

```
valMutex.lock();     // запрос на эксклюзивный доступ к переменной val
if (val >= 0) {
   f(val);           // val положительная
}
else {
   f(-val);          // передаем переменную val с противоположным знаком
}
valMutex.unlock();   // освобождаем эксклюзивный доступ к переменной val
```

Другой поток может обращаться к тому же самому ресурсу следующим образом:

```
valMutex.lock();   // запрос на эксклюзивный доступ к переменной val
++val;
valMutex.unlock(); // запрос на эксклюзивный доступ к переменной val
```

Следует подчеркнуть, что все места, в которых возможен конкурентный доступ, используют один и тот же мьютекс. Это относится как к доступу для чтения, так и к доступу для записи.

Однако этот простой подход может сильно усложниться. Например, необходимо гарантировать, что исключение, завершающее эксклюзивный доступ, разблокирует соответствующий мьютекс. В противном случае ресурс останется заблокированным навсегда. Кроме того, возможна взаимная блокировка, при которой один поток ожидает разблокировки другого потока, пока не освободится его собственная.

Стандартная библиотека C++ пытается решить эти проблемы, но в принципе не может этого сделать. Например, для решения проблемы, связанной с исключениями, нельзя блокировать и разблокировать мьютекс самостоятельно. В этой ситуации необходимо следовать принципу RAII (“*Resource Acquisition Is Initialization*” — “Захват ресурса — это инициализация”), согласно которому конструктор получает ресурс так, что деструктор, который вызывается всегда, даже когда существование объекта прекращает исключение, автоматически освобождает ресурс. Для этой цели в стандартную библиотеку C++ включен класс `std::lock_guard`:

```
int val;
std::mutex valMutex;       // управление эксклюзивным доступом к val
...
   std::lock_guard<std::mutex> lg(valMutex); // блокируем
                                             // и автоматически разблокируем
   if (val >= 0) {
       f(val);             // переменная val положительная
   }
   else {
       f(-val);  // передаем переменную val с противоположным знаком
}
```

Отметим, однако, что блокировки должны ограничиваться максимально коротким периодом времени, потому что они блокируют любое параллельное выполнение другого кода. Поскольку деструктор освобождает блокировку, можно явно вставить фигурные скобки, чтобы блокировка освобождалась до выполнения следующих инструкций:

```
int val;
std::mutex valMutex; // управление эксклюзивным доступом к val
...
{
   std::lock_guard<std::mutex> lg(valMutex); // блокируем
                                             // и автоматически разблокируем

   if (val >= 0) {
       f(val);      // переменная val положительная
   }
   else {
       f(-val);   // // передаем переменную val с противоположным знаком
   }
} // гарантируем, что блокировка освобождается здесь
...
```

или просто

```
...
{
   std::lock_guard<std::mutex> lg(valMutex); // блокируем
                                             // и автоматически разблокируем
   ++val;
} // гарантируем, что блокировка освобождается здесь
...
```

Это лишь первое приближение к решению, но уже можно убедиться, что ситуация постепенно усложняется. Как обычно, программисты должны помнить, что их программа выполняется в параллельном режиме. Кроме того, существуют разные мьютексы и блокировки, которые будут рассмотрены в следующих подразделах.

Первый полный пример использования мьютекса и блокировки

Рассмотрим первый полный пример:

```
// concurrency/mutex1.cpp

#include <future>
#include <mutex>
#include <iostream>
#include <string>

std::mutex printMutex;              // позволяет синхронизированный вывод
                                    // с помощью функции print()
void print (const std::string& s)
{
   std::lock_guard<std::mutex> l(printMutex);
   for (char c : s) {
        std::cout.put(c);
   }
   std::cout << std::endl;
}
```

```
int main()
{
  auto f1 = std::async (std::launch::async,
                        print, "Hello from a first thread");
  auto f2 = std::async (std::launch::async,
                        print, "Hello from a second thread");
  print("Hello from the main thread");
}
```

Здесь функция `print()` выводит все символы передаваемой строки в стандартный поток вывода. Таким образом, если бы не было блокировки, вывод мог выглядеть следующим образом[19]:

```
HHelHello from a second thread
ello from a first thread
lo from the main thread
```

или

```
HelloHello fHello from a second ro from am th fthe main irrethreadstad
 thr
ead
```

Для синхронизации вывода так, чтобы каждый вызов функции `print()` эксклюзивно выводил символы, мы ввели мьютекс для операции вывода и блокировку, которая защищает соответствующий раздел:

```
std::mutex printMutex; // позволяет синхронизированный вывод
                       // с помощью функции print()
...
void print (const std::string& s)
{
   std::lock_guard<std::mutex> l(printMutex);
   ...
}
```

Теперь вывод будет выглядеть так:

```
Hello from a first thread
Hello from the main thread
Hello from a second thread
```

Если бы блокировки не было, этот вывод был бы возможен (но не гарантирован).

Здесь функция мьютекса `lock()`, вызываемая конструктором блокировки, останавливает работу потока, если ресурс уже занят. Она блокирует поток до тех пор, пока доступ к защищенному разделу не откроется снова. Однако порядок блокировок остается неопределенным. Таким образом, три строки вывода могут быть записаны в произвольном порядке.

---

[19] Факт, что каждый символ записывается с помощью отдельного вызова функции `put()`, делает возможным появление промежуточных символов при параллельном выводе. Если бы строка выводилась целиком, то реализация, скорее всего, не перемешала бы символы, но это не гарантируется.

Рекурсивные блокировки

Иногда требуется возможность рекурсивной блокировки. Типичные примеры — активные объекты или мониторы, содержащие мьютексы и получающие блокировку в каждом открытом методе для защиты от конкуренции за данные, повреждающей внутреннее состояние объекта. Например, интерфейс базы данных может выглядеть следующим образом:

```
class DatabaseAccess
{
   private:
      std::mutex dbMutex;
      ... // состояние доступа к базе данных
   public:
      void createTable (...)
      {
         std::lock_guard<std::mutex> lg(dbMutex);
         ...
      }
      void insertData (...)
      {
         std::lock_guard<std::mutex> lg(dbMutex);
         ...
      }
      ...
};
```

Если ввести открытую функцию-член, которая может вызывать другую открытую функцию-член, ситуация может усложниться.

```
void createTableAndInsertData (...)
{
   std::lock_guard<std::mutex> lg(dbMutex);
   ...
   createTable(...); // ОШИБКА: взаимная блокировка, потому что
                     // dbMutex блокируется снова
}
```

Вызов функции `createTableAndInsertData()` приводит к взаимной блокировке, потому что после блокирования `dbMutex` вызов функции `createTable()` попытается снова заблокировать `dbMutex`, пока его блокировка не станет доступной, а это никогда не произойдет, так как функция `createTableAndInsertData()` блокирует поток до тех пор, пока не будет вызвана функция `createTable()`.

Стандартная библиотека C++ допускает вторую попытку генерирования исключения `std::system_error` (см. раздел 4.3.1) с кодом ошибки `resource_deadlock_would_occur` (см. раздел 4.3.2), если платформа распознает такую взаимную блокировку. Однако это не обязательное требование, и обычно оно не выполняется.

Использование мьютекса `recursive_mutex` решает описанную проблему. Этот мьютекс допускает многократные блокировки одного и того же потока и освобождает блокировку, когда происходит последний вызов функции `unlock()`.

```
class DatabaseAccess
{
   private:
      std::recursive_mutex dbMutex;
      ... // состояние доступа к базе данных
   public:
      void insertData (...)
      {
         std::lock_guard<std::recursive_mutex> lg(dbMutex);
         ...
      }
      void insertData (...)
      {
         std::lock_guard<std::recursive_mutex> lg(dbMutex);
         ...
      }
      void createTableAndinsertData (...)
      {
         std::lock_guard<std::recursive_mutex> lg(dbMutex);
         ...
         createTable(...); // OK: взаимной блокировки нет
      }
     ...
};
```

Попытки блокировки и временные блокировки

Иногда программа хочет установить блокировку, но не хочет блокировать мьютекс (навсегда), если это невозможно. В такой ситуации мьютексы предоставляют функцию-член `try_lock()`, которая *пытается* установить блокировку. Если это удается, она возвращает значение `true`; если нет — `false`.

Для того чтобы сохранить возможность использовать объект класса `lock_guard`, т.е. при любом выходе из текущей области видимости автоматически разблокировать мьютекс, можно передать его конструктору дополнительный аргумент `adopt_lock`.

```
std::mutex m;

// пытаемся овладеть блокировкой и сделать другую работу, пока это возможно
while (m.try_lock() == false) {
   doSomeOtherStuff();
}
std::lock_guard<std::mutex> lg(m,std::adopt_lock);
...
```

Отметим, что функция `try_lock()` может отказать по непонятным причинам. Иначе говоря, она может отказать (вернуть значение `false`), даже если блокировка свободна[20].

Для того чтобы ожидание не превышало установленного периода времени, можно использовать временной мьютекс. Специальные классы мьютекса `std::timed_mutex` и `std::recursive_timed_mutex` дополнительно позволяют вызывать функцию

[20] Это поведение связано с особенностями упорядочения памяти и мало известно. Благодарю Ганса Боэма и Бартоша Милевски за это указание.

try_lock_for() или try_lock_until(), для того чтобы ожидание не превышало указанного периода времени или пока не настанет определенный момент времени. Это, например, может помочь избежать взаимных блокировок и учесть требования систем реального времени. Рассмотрим пример:

```
std::timed_mutex m;

// пытаемся установить блокировку в течение одной секунды
if (m.try_lock_for(std::chrono::seconds(1))) {
   std::lock_guard<std::timed_mutex> lg(m,std::adopt_lock);
   ...
}
else {
   couldNotGetTheLock();
}
```

Отметим, что обычно функции try_lock_for() и try_lock_until() отличаются способами корректировки системного времени (см. раздел 5.7.5).

Работа с несколькими блокировками

Обычно в один момент времени поток блокирует только один мьютекс. Однако иногда необходимо заблокировать несколько мьютексов (например, для передачи от одного защищенного ресурса другому). В этом случае работа с механизмами блокировки, описанная выше, усложняется и становится рискованной. Можно установить первую блокировку, но не вторую, или может возникнуть взаимная блокировка, если одни и те же блокировки устанавливаются в разном порядке.

По этим причинам стандартная библиотека C++ содержит вспомогательные функции для работы с несколькими мьютексами. Рассмотрим пример:

```
std::mutex m1;
std::mutex m2;
...
{
   std::lock (m1, m2); // блокируем оба мьютекса (или ни один из них,
                       // если это невозможно)
   std::lock_guard<std::mutex> lockM1(m1,std::adopt_lock);
   std::lock_guard<std::mutex> lockM2(m2,std::adopt_lock);
   ...
} //автоматически разблокируем все мьютексы
```

Глобальная функция std::lock() блокирует все мьютексы, передаваемые как аргументы, до тех пор, пока они не будут разблокированы или не будет сгенерировано исключение. В последнем случае она разблокирует мьютексы, которые были успешно заблокированы. Как обычно, после успешной блокировки можно и нужно использовать защиту блокировки (lock guard), инициализированную объектом класса adopt_lock, передаваемым как второй аргумент. Это позволит разблокировать мьютекс при выходе из области видимости. Функция lock() предотвращает взаимные блокировки, хотя порядок блокировки нескольких мьютексов не определен.

Точно так же можно *попытаться* выполнить блокировку несколькими мьютексами, не блокируя поток, если окажется неудачной хотя бы одна попытка. Глобальная функция

`std::try_lock()` возвращает -1, если все блокировки доступны. Если нет, возвращаемое значение равно индексу первой неудачной блокировки (при этом отсчет индексов начинается с нуля). В этом случае все успешные блокировки снова снимаются.

```
std::mutex m1;
std::mutex m2;

int idx = std::try_lock (m1, m2); // пытаемся заблокировать оба мьютекса
if (idx < 0) { // обе блокировки установлены успешно
   std::lock_guard<std::mutex> lockM1(m1,std::adopt_lock);
   std::lock_guard<std::mutex> lockM2(m2,std::adopt_lock);
   ...
} //автоматически разблокируем все мьютексы
else {
   // idx содержит индекс первой неудачной блокировки, считая с нуля
std::cerr << "could not lock mutex m" << idx+1 << std::endl;
}
```

Функция `try_lock()` не позволяет предотвратить взаимную блокировку. Вместо этого она гарантирует, что попытки блокирования будут выполнены в порядке передачи аргументов.

Кроме того, вызов функций `lock()` и `try_lock()` без адаптации блокировок с помощью защиты обычно нецелесообразен. Несмотря на то что на первый взгляд код создает блокировки, которые автоматически снимаются при выходе из области видимости, это не так. Мьютексы остаются заблокированными.

```
std::mutex m1;
std::mutex m2;
...
{
   std::lock (m1, m2); // блокируем оба мьютекса (или ни одного,
                       // если это невозможно)
   // ни одна блокировка не принята
   ...
}
... // ОЙ: мьютексы остаются заблокированными!!!
```

Класс unique_lock

Помимо класса `lock_guard<>`, стандартная библиотека C++ содержит класс `unique_lock<>`, использование которого намного гибче, чем работа с блокировками мьютексов. Класс `unique_lock<>` предоставляет тот же самый интерфейс, что и класс `lock_guard<>`, и дополняет его возможностью явного программирования, когда и как блокировать и разблокировать мьютекс. Таким образом, этот объект блокировки может иметь как заблокированный, так и не заблокированный мьютекс (это состояние называется *владением* мьютексом). Этим класс `unique_lock<>` отличается от класса `lock_guard<>`, объект которого всегда заблокирован на протяжении всего существования[21]. Кроме того,

[21] Слово *unique* в имени класса описывает истоки его поведения. Аналогично уникальным указателям (см. раздел 5.2.5), вы можете передавать мьютексы за пределы области видимости, но гарантируется только, что в каждый момент времени мьютексом владеет только одна блокировка.

уникальным блокировкам можно передавать запрос об индексе, является ли текущий индекс заблокированным, вызвав функцию `owns_lock()` или `operator bool()`.

Основное преимущество этого класса заключается в том, что если мьютекс оказывается заблокированным в момент его уничтожения, то деструктор автоматически применяет к нему функцию `unlock()`. Если мьютекс не заблокирован, деструктор ничего не делает.

По сравнению с классом `lock_guard` класс `unique_lock` имеет следующие дополнительные конструкторы.

- Можно передать конструктору объект класса `try_to_lock` для выполнения попытки заблокировать мьютекс без блокирования потока.

```
std::unique_lock<std::mutex> lock(mutex,std::try_to_lock);
...
if (lock) { // если блокировка была успешной
...
}
```

- Можно передать конструктору продолжительность или момент времени, до которого следует заблокировать мьютекс.

```
std::unique_lock<std::timed_mutex> lock(mutex,
std::chrono::seconds(1));
...
```

- Можно передать конструктору флаг `defer_lock` для инициализации объекта блокировки без фактической блокировки мьютекса (отложенная блокировка).

```
std::unique_lock<std::mutex> lock(mutex,std::defer_lock);
...
lock.lock(); // или (временной) try_lock()
...
```

Флаг `defer_lock` может, например, использоваться для создания одной или нескольких блокировок и их дальнейшей установки.

```
std::mutex m1;
std::mutex m2;
std::unique_lock<std::mutex> lockM1(m1,std::defer_lock);
std::unique_lock<std::mutex> lockM2(m2,std::defer_lock);
...
std::lock (m1, m2); // блокируем оба мьютекса (или ни один, если это невозможно)
```

Кроме того, класс `unique_lock` предоставляет возможность применить к мьютексу функцию `release()` или передать владение мьютексом другой блокировке (см. раздел 18.5.2).

Классы `lock_guard` и `unique_lock` позволяют реализовать наивный пример, в котором один поток ждет другого, опрашивая *флаг готовности*.

```
#include <mutex>
...
bool readyFlag;
std::mutex readyFlagMutex;

void thread1()
```

```
{
   // готовим поток thread2
   ...
   std::lock_guard<std::mutex> lg(readyFlagMutex);
   readyFlag = true;
}

void thread2()
{
   // ждем, пока флаг readyFlag станет равным true (поток thread1 готов)
   {
      std::unique_lock<std::mutex> ul(readyFlagMutex);
      while (!readyFlag) {
         ul.unlock();
         std::this_thread::yield(); // подсказка для изменения момента
                                    // запуска следующего потока
         std::this_thread::sleep_for(std::chrono::milliseconds(100));
         ul.lock();
   }
} // освобождает блокировку

// продолжаем работу после выполнения потока thread1
...
}
```

Этот код обычно вызывает два вопроса.

- Почему мы используем мьютекс для управления доступом для чтения и записи флага `readyFlag`? Вспомните правило, сформулированное в начале главы: любой конкурентный доступ, предусматривающий хотя бы одну операцию записи, должен быть синхронизирован (см. разделы 18.4 и 18.7).
- Почему не использовано ключевое слово `volatile` для объявления флага `readyFlag`, чтобы избежать оптимизации многочисленных попыток функции `thread2()` прочитать его? Дело в том, что попытки прочитать флаг `readyFlag` осуществляются в *критическом разделе*, определенном между моментами установки и снятия блокировки. Такой код не может оптимизироваться путем переноса операций чтения (или записи) за пределы критического раздела. Таким образом, чтение флага `readyFlag` должно быть реализовано эффективно в следующих местах.
  - В начале цикла между объявлением объекта `ul` и первым вызовом функции `unlock()`.
  - В цикле между вызовами функций `lock()` и `unlock()`.
  - В конце цикла между последним вызовом функции `lock()` и уничтожением объекта `ul`, которое разблокирует мьютекс, если он был заблокирован.

Тем не менее такой *опрос* условия обычно не является хорошим решением. В таких ситуациях лучше использовать *условные переменные* (подробно об этом — в разделе 18.6.1).

18.5.2. Подробное описание мьютексов и блокировок

Подробное описание мьютексов

Стандартная библиотека C++ содержит следующие классы мьютексов (см. раздел 18.6).

- Класс `std::mutex` — простой мьютекс, который в каждый момент времени может быть заблокирован только одним потоком. Если он заблокирован, то любой другой вызов функции `lock()` блокирует остальные потоки до тех пор, пока мьютекс снова не станет доступным и функция `try_lock()` не вернет значение `false`.
- Класс `std::recursive_mutex` — мьютекс, позволяющий одному потоку устанавливать несколько одновременных блокировок. Типичное применение такого мьютекса — ситуация, в которой функции устанавливают блокировку и вызывают другие функции, которые снова устанавливают ту же самую блокировку.

Таблица 18.6. Обзор мьютексов и их возможностей

Операция	`mutex`	`recursive_mutex`	`timed_mutex`	`recursive_timed_mutex`
`lock()`	Овладевает мьютексом (блокирует поток, если он не доступен)			
`try_lock()`	Овладевает мьютексом (возвращает значение `false`, если мьютекс не доступен)			
`unlock()`	Разблокирует заблокированный мьютекс			
`try_lock_for()`	–	–	Пытается овладеть блокировкой на определенный период времени	
`try_lock_until()`	–	–	Пытается овладеть блокировкой до определенного момента времени	
Несколько блокировок	Нет	Да (тот же поток)	Нет	Да (тот же поток)

- Класс `std::timed_mutex` — простой мьютекс, который дополнительно позволяет передавать продолжительность или момент времени для ограничения попыток овладеть блокировкой. Для этой цели предназначены функции `try_lock_for()` и `try_lock_until()`.
- Класс `std::recursive_timed_mutex` — мьютекс, допускающий несколько блокировок, установленных одним и тем же потоком с необязательными временными параметрами.

Операции над мьютексами приведены в табл. 18.7

Таблица 18.7. Операции над мьютексами

Операция	Описание
mutex m	Конструктор по умолчанию. Создает незаблокированный мьютекс
m.~***mutex***()	Уничтожает мьютекс (который не должен быть заблокированным)
m.`lock()`	Блокирует мьютекс (если мьютекс уже был заблокирован и не является рекурсивным, возникает ошибка)

Окончание табл. 18.7

Операция	Описание
m.try_lock()	Пытается заблокировать мьютекс (возвращает true, если блокировка была успешной)
m.try_lock_for(*dur*)	Пытается заблокировать мьютекс в течение периода *dur* (если блокировка была успешной, возвращает true)
m.try_lock_until(*tp*)	Пытается заблокировать мьютекс до момента *tp* (если блокировка была успешной, возвращает true)
m.unlock()	Разблокирует мьютекс (если мьютекс не был заблокирован, то это приведет к непредвиденным последствиям)
m.native_handle()	Возвращает объект типа native_handle_type для непереносимых расширений, зависящих от платформы

Функция lock() может генерировать исключение std::system_error (см. раздел 4.3.1) со следующими кодами ошибок (см. раздел 4.3.2):

- operation_not_permitted, если поток не имеет права выполнять данную операцию;
- resource_deadlock_would_occur, если платформа обнаруживает возможность взаимной блокировки;
- device_or_resource_busy, если мьютекс уже заблокирован и блокировка невозможна.

Поведение программы не определено, если она разблокирует объект мьютекса, которым не владеет, уничтожает объект мьютекса, которым владеет какой-то поток, или выполнение потока прекращается после овладения объектом мьютекса.

Отметим, что функции try_lock_for() и try_lock_until() обычно отличаются работой с настройками системного времени (см. раздел 5.7.5).

Подробное описание класса lock_guard

Класс std::lock_guard, введенный в разделе 18.5.1, имеет очень небольшой интерфейс, гарантирующий, что заблокированный мьютекс всегда будет освобождаться после выхода из области видимости (см. табл. 18.8). На протяжении времени своего существования он всегда ассоциируется с блокировкой, которая запрашивается явно или устанавливается во время создания объекта.

Таблица 18.8. Операции над классом lock_guard

Операция	Описание
lock_guard *lg*(*m*)	Создает защиту блокировки для мьютекса *m* и блокирует его
lock_guard *lg*(*m*, adopt_lock)	Создает защиту блокировки для уже заблокированного мьютекса *m*
lg.~***lock_guard***()	Разблокирует мьютекс и уничтожает защиту блокировки

Подробное описание класса unique_lock

Класс std::unique_lock, введенный в разделе 18.5.1, создает защиту блокировки для мьютекса, который не обязательно блокировать (владеть им). Его интерфейс показан в табл. 18.9.

Если он блокирует мьютекс (вступает во владение им) в момент уничтожения, мьютекс разблокируется с помощью функции unlock(). Однако можно явно проверить, существует ли ассоциированный мьютекс и заблокирован ли он. Мьютекс можно попытаться блокировать как с задержками, так и без них.

Функция lock() может генерировать исключение std::system_error (см. раздел 4.3.1) с кодами ошибок, перечисленными при описании функции lock() для мьютексов (см. табл. 18.7). Функция unlock() может генерировать исключение std::system_error с кодом ошибки operation_not_permitted, если уникальная блокировка не была установлена.

Таблица 18.9. Операции над классом unique_guard

Операция	Описание
unique_lock *l*	Конструктор по умолчанию. Создает блокировку, не ассоциированную с мьютексом
unique_lock *l*(*m*)	Создает защиту блокировки для мьютекса *m* и блокирует его
unique_lock *l*(*m*, adopt_lock)	Создает защиту блокировки для уже заблокированного мьютекса *m*
unique_lock *l*(*m*, defer_lock)	Создает защиту блокировки для мьютекса *m* без его блокирования
unique_lock *l*(*m*, try_lock)	Создает защиту блокировки для мьютекса *m* и пытается блокировать его
unique_lock *l*(*m*, *dur*)	Создает защиту блокировки для мьютекса *m* и пытается блокировать его на период времени *dur*
unique_lock *l*(*m*, *tp*)	Создает защиту блокировки для мьютекса *m* и пытается блокировать его до момента *tp*
unique_lock *l*(*rv*)	Перемещающий конструктор. Переносить состояние блокировки с объекта *rv* на объект *l* (объект *rv* больше не имеет ассоциированного мьютекса)
l.~***unique_lock***()	Разблокирует мьютекс, если он был заблокирован, и уничтожает защиту блокировки
unique_lock *l* = *rv*	Перемещающее присваивание. Перемещает состояние блокировки с объекта *rv* на объект *l* (объект *rv* больше не имеет ассоциированного мьютекса)
swap(*l1*, *l2*)	Обменивает блокировки
l1.swap(*l2*)	Обменивает блокировки
l.release()	Возвращает указатель на ассоциированный мьютекс и освобождает его
l.owns_lock()	Возвращает true, если ассоциированный мьютекс заблокирован

Окончание табл. 18.9

Операция	Описание
`if (`*l*`)`	Проверяет, заблокирован ли ассоциированный мьютекс
l`.mutex()`	Возвращает указатель на ассоциированный мьютекс
l`.lock()`	Блокирует ассоциированный мьютекс
l`.try_lock()`	Пытается заблокировать ассоциированный мьютекс (возвращает `true`, если блокировка была успешной)
l`.try_lock_for(`*dur*`)`	Пытается заблокировать ассоциированный мьютекс на период времени *dur* (возвращает `true`, если блокировка была успешной)
l`.try_lock_until(`*tp*`)`	Пытается заблокировать ассоциированный мьютекс до момента *tp* (возвращает `true`, если блокировка была успешной)
l`.unlock()`	Разблокирует ассоциированный мьютекс

18.5.3. Одновременный вызов нескольких потоков

Иногда несколько потоков не должны выполнять действия, которые выполнены первым потоком. Типичным примером является "ленивая инициализация": действия выполняются, только когда от некоторого потока требуется их выполнение, и не ранее, потому что вы хотите сэкономить время, не выполняя действия, которые будут не нужны.

В однопоточной среде эта задача решается просто: булев флаг сигнализирует, была ли вызвана функция:

```
bool initialized = false; // глобальный флаг
...
if (!initialized) {       // выполняем инициализацию,
    initialize();         // если она не была выполнена ранее
    initialized = true;
}
```

или

```
static std::vector<std::string> staticData;

void foo()
{
   if (staticData.empty()) {
       staticData = initializeStaticData();
   }
   ...
}
```

Но этот код не работает в многопоточном контексте, потому что, если несколько потоков станут проверять, была ли выполнена инициализация, и начнут ее выполнять, может возникнуть состязание за данные. Таким образом, необходимо защитить область проверки и инициализации от конкурентного доступа.

Как обычно, для этого можно использовать мьютексы, но стандартная библиотека C++ имеет специальное решение для этой задачи. Можно просто использовать флаг `std::once_flag` и вызвать функцию `std::call_once` (также определенную в заголовочном файле <mutex>).

```
std::once_flag oc;              // глобальный флаг
...
std::call_once(oc,initialize); // выполняем инициализацию,
                               // если она не была выполнена ранее
```

или

```
static std::vector<std::string> staticData;

void foo()
{
   static std::once_flag oc;
   std::call_once(oc,[]{
                        staticData = initializeStaticData();
                      });
   ...
}
```

Как видим, первый аргумент, передаваемый функции `call_once()`, должен быть соответствующим флагом `once_flag`. Остальные аргументы вполне типичны для *вызываемых объектов*: функций, функций-членов, функциональных объектов или лямбда-функций. Кроме того, существуют необязательные аргументы (см. раздел 4.4). Таким образом, ленивая инициализация объекта в многопоточной среде может выглядеть следующим образом:

```
class X {
   private:
      mutable std::once_flag initDataFlag;
      void initData() const;
   public:
      data getData () const {
        std::call_once(initDataFlag,&X::initData,this);
        ...
   }
};
```

В принципе с одним и тем же флагом `once_flag` можно вызывать несколько функций. Этот флаг передается функции `call_once()` как первый аргумент и гарантирует, что передаваемая функциональная сущность будет выполняться только один раз. Итак, если первый вызов был успешным, то последующие вызовы с тем же флагом не будут приводить к выполнению передаваемой функциональной сущности, даже если она отличается от первой.

Любое исключение, созданное вызываемой функциональной сущностью, также генерируется функцией `call_once()`. В этом случае первый вызов считается неудачным, так что следующий вызов функции `call_once()` будет по-прежнему выполнять передаваемую функциональную сущность[22].

[22] Стандарт также указывает, что функция `call_once()` может генерировать исключение `std::system_error`, если аргумент `once_flag` больше не является корректным (т.е. уничтожен). Однако это утверждение ошибочно, потому что передача уничтоженного флага либо невозможна, либо приводит к непредсказуемым последствиям.

18.6. Условные переменные

Иногда задачи, выполняемые разными потоками, должны ожидать друг друга. Следовательно, иногда возникает необходимость синхронизировать параллельные операции не только из-за доступа к одним и тем же данным.

Может показаться, что мы уже описывали этот механизм: фьючерсы (см. раздел 18.1) позволяют блокировать поток, пока не поступят данные от другого потока или пока другой поток не будет завершен. Однако фьючерс может передавать данные от одного потока другому только один раз. По существу, основная цель фьючерса — обработка значений или исключений, возвращаемых потоками.

В этом разделе рассматриваются условные переменные, которые можно использовать для синхронизации логических зависимостей, существующих в потоках данных, которыми многократно обмениваются несколько потоков.

18.6.1. Предназначение условных переменных

В разделе 18.5.1 описан наивный подход, позволяющий одному потоку ожидать другого с помощью *флага готовности*, сигнализирующего о том, что поток готов к выполнению или подготовил данные для другого потока. Обычно это означает, что ожидающий поток *опрашивает* другой поток в ожидании требуемых данных или проверяет предусловие:

```
bool readyFlag;
std::mutex readyFlagMutex;

// ждем, пока readyFlag равен true:
{
   std::unique_lock<std::mutex> ul(readyFlagMutex);
   while (!readyFlag) {
      ul.unlock();
      std::this_thread::yield(); // подсказка для запуска следующего потока
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
                                 ul.lock();
   }
} // освобождаем блокировку
```

Однако такой *опрос* для проверки условий обычно не является удачным решением. Как указано в работе [*Williams:C++Conc*]:

"Ожидающий поток затрачивает ценное машинное время, многократно проверяя флаг, и когда он блокирует мьютекс, поток, установивший флаг, становится заблокированным... Кроме того, трудно правильно вычислить период простоя: слишком короткий период между проверками приводит к потерям машинного времени, а слишком длинный период приводит к тому, что поток продолжает простаивать даже тогда, когда задача, которую он ожидал, уже выполнена.

Много лучше использовать *условные переменные*, которые стандартная библиотека C++ описывает в заголовочном файле `<condition_variable>`. Условная переменная — это переменная, с помощью которой поток может активизировать один или несколько других ожидающих потоков.

В принципе условная переменная работает следующим образом.

- Необходимо включить в программу заголовочные файлы <mutex> и <condition_variable>, чтобы объявить мьютекс и условную переменную.

```
#include <mutex>
#include <condition_variable>

std::mutex readyMutex;
std::condition_variable readyCondVar;
```

- Поток (или несколько потоков), сигнализирующий о выполнении условия, должен выполнить вызов

```
readyCondVar.notify_one(); // уведомляем один из ожидающих потоков
```

или

```
readyCondVar.notify_all(); // уведомляем все ожидающие потоки
```

- Любой поток, ожидающий выполнения условия, должен выполнить вызов

```
std::unique_lock<std::mutex> l(readyMutex);
readyCondVar.wait(l);
```

Таким образом, поток, предоставляющий или подготавливающий данные для других потоков, иногда просто вызывает функции `notify_one()` или `notify_all()` из условной переменной, и в этот момент активизируется один или несколько ожидающих потоков.

До сих пор все было хорошо и просто. Однако есть еще кое-что. Во-первых, для ожидания условной переменной нужен мьютекс и объект класса `unique_lock`, введенный в разделе 18.5.1. Объекта класса `lock_guard` недостаточно, потому что ожидающая функция может блокировать и разблокировать мьютекс. Во-вторых, условные переменные теоретически могут вызывать *ложные срабатывания* (spurious wakeups). Иначе говоря, проверка условной переменной может сработать, даже если условная переменная не осуществляла уведомление. Процитируем Энтони Уильямса (Anthony Williams) [*Williams:CondVar*]: "Ложные срабатывания невозможно предсказать: с точки зрения пользователя они являются совершенно случайными. Однако они часто происходят, когда библиотека потоков не может гарантировать, что ожидающий поток не пропустит уведомление. Поскольку пропущенное уведомление делает условную переменную бесполезной, библиотека потоков активизирует потоки, чтобы не рисковать".

Итак, активизация не всегда означает, что требуемое условие выполнено. После активизации необходимо еще раз проверить выполнение условия. Следовательно, необходимо проверить, например, действительно ли доступны передаваемые данные и установлен ли флаг готовности. Для того чтобы проверить эти условия, можно использовать тот же самый мьютекс.

18.6.2. Первый законченный пример использования условных переменных

Следующая программа представляет собой законченный пример, демонстрирующий использование условных переменных:

```
// concurrency/condvar1.cpp

#include <condition_variable>
```

```
#include <mutex>
#include <future>
#include <iostream>
bool readyFlag;
std::mutex readyMutex;
std::condition_variable readyCondVar;

void thread1()
{
   // подготовка данных для потока thread2
   std::cout << "<return>" << std::endl;
   std::cin.get();

   // сигнал, что поток thread1 выполнил условие
   {
      std::lock_guard<std::mutex> lg(readyMutex);
      readyFlag = true;
   } // освобождаем блокировку
   readyCondVar.notify_one();
}

void thread2()
{
   // ожидаем, пока поток thread1 не перейдет в состояние готовности
   // (флаг readyFlag равен true)
   {
      std::unique_lock<std::mutex> ul(readyMutex);
      readyCondVar.wait(ul, []{ return readyFlag; });
   } // освобождаем блокировку

   // выполняем операции, которые должны быть выполнены после того, как
   // поток thread1 закончит подготовку
   std::cout << "done" << std::endl;
}

int main()
{
   auto f1 = std::async(std::launch::async,thread1);
   auto f2 = std::async(std::launch::async,thread2);
}
```

После включения необходимых заголовочных файлов необходимы три вещи для обеспечения связи между тремя потоками.

1. Объект для хранения подготавливаемых данных или флага, сигнализирующего о действительном выполнении условия (в данном случае `readyFlag`)
2. Мьютекс (в данном случае `readyMutex`)
3. Условная переменная (в данном случае `readyCondVar`)

Поток `thread1()`, передающий данные, блокирует мьютекс `readyMutex`, обновляет условие (объект для данных или флаг готовности), разблокирует мьютекс и уведомляет условную переменную.

```
{
   std::lock_guard<std::mutex> lg(readyMutex);
   readyFlag = true;
} // освобождаем блокировку
readyCondVar.notify_one();
```

Отметим, что уведомление само по себе не должно быть в защищенной области блокировки.

Ожидающий (получающий/обрабатывающий данные) поток блокирует мьютекс с помощью объекта `unique_lock` (раздел 18.5.1), ожидает уведомления, проверяя условие, и освобождает блокировку:

```
{
   std::unique_lock<std::mutex> ul(readyMutex);
   readyCondVar.wait(ul, []{ return readyFlag; });
} // освобождаем блокировку
```

Здесь функция-член `wait()` условной переменной используется следующим образом: ей передается блокировка `ul` для мьютекса `readyMutex` в качестве первого аргумента и лямбда-функция как *вызываемый объект* (см. раздел 4.4), дважды проверяющий условие, в качестве второго аргумента. Эффект заключается в том, что функция `wait()` выполняет цикл, пока переданный вызываемый объект не вернет значение `true`. Таким образом, наш код эквивалентен следующему коду, в котором явно указан цикл, необходимый для обработки ложных срабатываний:

```
{
   std::unique_lock<std::mutex> ul(readyMutex);
   while (!readyFlag) {
          readyCondVar.wait(ul);
   }
} // освобождаем блокировку
```

Снова подчеркнем, что здесь необходимо использовать класс `unique_lock`, а не `lock_guard`, потому что функция `wait()` явно разблокирует и блокирует мьютекс.

Можно возразить, что этот пример плохо иллюстрирует возможности условных переменных, потому что для блокировки потока до получения данных можно использовать фьючерсы. Рассмотрим второй пример.

18.6.3. Использование условных переменных для реализации очереди для нескольких потоков

В этом примере три потока заталкивают значения в очередь, которую читают и обрабатывают два других потока.

```
// concurrency/condvar2.cpp

#include <condition_variable>
#include <mutex>
#include <future>
#include <thread>
#include <iostream>
```

```
#include <queue>
std::queue<int> queue;
std::mutex queueMutex;
std::condition_variable queueCondVar;

void provider (int val)
{
   // заталкиваем в очередь разные значения (от val до val+5 с задержками,
   // равными val миллисекунд
   for (int i=0; i<6; ++i) {
      {
         std::lock_guard<std::mutex> lg(queueMutex);
         queue.push(val+i);
      } // освобождаем блокировку
      queueCondVar.notify_one();

      std::this_thread::sleep_for(std::chrono::milliseconds(val));
   }
}

void consumer (int num)
{
   // выталкиваем из очереди значения, если они доступны
   // (число num обозначает получателя)
   while (true) {
      int val;
     {
          std::unique_lock<std::mutex> ul(queueMutex);
          queueCondVar.wait(ul,[]{ return !queue.empty(); });
          val = queue.front();
          queue.pop();
     } // освобождаем блокировку
     std::cout << "consumer " << num << ": " << val << std::endl;
   }
}

int main()
{
   // запускаем три потока-поставщика значений 100+, 300+ и 500+
   auto p1 = std::async(std::launch::async,provider,100);
   auto p2 = std::async(std::launch::async,provider,300);
   auto p3 = std::async(std::launch::async,provider,500);

   // запускам два потока-получателя, которые выводят значения
   auto c1 = std::async(std::launch::async,consumer,1);
   auto c2 = std::async(std::launch::async,consumer,2);
}
```

Здесь есть глобальная очередь (см. раздел 12.2), используемая несколькими потоками и защищенная мьютексом и условной переменной:

```
std::queue<int> queue;
std::mutex queueMutex;
std::condition_variable queueCondVar;
```

Мьютекс гарантирует, что чтение и запись являются атомарными операциями, а условная переменная используется для сигнализации и активизации обрабатывающих потоков, при условии, что доступны новые значения.

Три потока вносят данные в очередь, делая их доступными для других потоков:

```
{
   std::lock_guard<std::mutex> lg(queueMutex);
   queue.push(val+i);
} // особождаем блокировку
queueCondVar.notify_one();
```

Используя функцию `notify_one()`, эти потоки активизируют ожидающие потоки для обработки следующего значения. Подчеркнем, что этот вызов не обязан быть частью защищенного раздела, поэтому мы разместили этот блок после закрытия блока, в котором объявлена защита блокировки.

Потоки, ожидающие новые значения, работают следующим образом:

```
int val;
{
   std::unique_lock<std::mutex> ul(queueMutex);
   queueCondVar.wait(ul,[]{ return !queue.empty(); });
   val = queue.front();
   queue.pop();
} // освобождаем блокировку
...
```

В соответствии с интерфейсом очереди (см. раздел 12.2) мы должны вызвать три функции, чтобы вытолкнуть из очереди следующее значение. Функция `empty()` проверяет, доступно ли значение. Вызов функции `empty()` выполняется дважды, чтобы предотвратить ложное срабатывание в функции `wait()`. Функция `front()` запрашивает следующее значение, а функция `pop()` удаляет его. Все три функции находятся в защищенной области уникальной блокировки `ul`. Однако обработка значения, возвращенного функцией `front()`, выполняется позднее, чтобы минимизировать период блокировки.

Возможный вывод программы выглядит следующим образом:

```
consumer 1: 300
consumer 1: 100
consumer 2: 500
consumer 1: 101
consumer 2: 102
consumer 1: 301
consumer 2: 103
consumer 1: 104
consumer consumer 1: 105
2: 501
consumer 1: 302
consumer 2: 303
consumer 1: 502
consumer 2: 304
consumer 1: 503
consumer 2: 305
consumer 1: 504
consumer 2: 505
```

Вывод двух потоков-получателей не синхронизирован, поэтому возможны перемежающиеся символы. Кроме того, порядок уведомления ожидающих потоков также не определен.

Точно так же можно вызвать функцию `notify_all()`, если несколько получателей должны обрабатывать одни и те же данные. Типичным примером может быть событийно-управляемая система, в которой событие должно быть опубликовано для всех зарегистрированных получателей.

Отметим также, что условные переменные имеют интерфейс для ожидания с минимальным количеством времени: функция `wait_for()` обеспечивает ожидание на протяжении заданного периода времени, а функция `wait_until()` ожидает наступления заданного момента времени.

18.6.4. Подробное описание условных переменных

Заголовочный файл `<condition_variable>` содержит два класса условных переменных: `condition_variable` и `condition_variable_any`.

Класс `condition_variable`

Как указано в разделе 18.6, класс `std::condition_variable`, содержащийся в библиотеке C++, позволяет активизировать один или несколько потоков, ожидающих выполнения определенного условия (выполнения необходимого действия или подготовки требуемых данных). Одну и ту же условную переменную могут ожидать несколько потоков. Когда условие выполнено, поток может уведомить об этом один или несколько ожидающих потоков.

Из-за *ложных срабатываний* уведомление потока еще не означает, что условие действительно выполнено. Ожидающий поток должен иметь возможность двойной проверки выполнения условия после своей активизации.

В табл. 18.10 подробно описан интерфейс стандартной библиотеки C++ для класса `condition_variable`. Класс `condition_variable_any` имеет аналогичный интерфейс, за исключением функций `native_handle()` и `notify_all_at_thread_exit()`.

Если не удается создать условную переменную, конструктор может сгенерировать исключение `std::system_error` (см. раздел 4.3.1) с кодом ошибки `resource_unavailable_try_again`, который эквивалентен ошибке `errno` `EAGAIN` в POSIX-системах (см. раздел 4.3.2). Копирование и присваивание не разрешаются.

Таблица 18.10. Операции над классом `condition_variable`

Операция	Описание
condvar *cv*	Конструктор по умолчанию. Создает условную переменную
cv.~***condvar***()	Уничтожает условную переменную
cv.`notify_one()`	Активизирует один из ожидающих потоков, если таковой имеется
cv.`notify_all()`	Активизирует все ожидающие потоки
cv.`wait(`***ul***`)`	Ожидает уведомления, используя уникальную блокировку *ul*

Окончание табл. 18.10

Операция	Описание
cv.wait(*ul*,*pred*)	Ожидает уведомления, используя уникальную блокировку *ul*, пока предикат *pred* не вернет значение true после активизации
cv.wait_for(*ul*,*duration*)	Ожидает уведомления, используя уникальную блокировку *ul* в течение периода времени *duration*
cv.wait_for(*ul*,*duration*,*pred*)	Ожидает уведомления, используя уникальную блокировку *ul* в течение периода времени *duration* или пока предикат *pred* не вернет значение true после активизации
cv.wait_until(*ul*,*timepoint*)	Ожидает уведомления, используя уникальную блокировку *ul* до момента времени *timepoint*
cv.wait_until(*ul*,*timepoint*,*pred*)	Ожидает уведомления, используя уникальную блокировку *ul* до момента времени *timepoint* или пока предикат *pred* не вернет значение true после активизации
cv.native_handle()	Возвращает объект платформозависимого типа native_handle_type для непереносимых исключений
notify_all_at_thread_exit(*cv*,*ul*)	Активизирует все потоки, ожидающие условную переменную *cv*, используя уникальную блокировку *ul* в конце вызывающего потока

Уведомления автоматически синхронизируются, чтобы параллельные вызовы функций notify_one() и notify_all() не создавали проблем.

Все потоки, ожидающие условную переменную, должны использовать один и тот же мьютекс, который должен быть заблокирован объектом класса unique_lock при вызове одной из функций-членов wait(). В противном случае возникают непредвиденные последствия.

Отметим, что потоки, ожидающие условную переменную, всегда оперируют мьютексами, которые обычно заблокированы. Только ожидающие функции временно разблокируют мьютекс, выполняя три атомарные операции[23].

1. Разблокирование мьютекса и вход в состояние ожидания.
2. Разблокирование ожидающего потока.
3. Повторное блокирование мьютекса.

Это значит, что предикаты, передаваемые ожидающим функциям, всегда вызываются под блокировкой, чтобы они могли безопасно обращаться к объекту или объектам, защищенным мьютексом[24]. Вызовы блокировки и разблокировки мьютекса могут привести к появлению соответствующих исключений (см. раздел 18.5.2).

[23] Проблема, связанная с наивным подходом, который сводится к формуле "*блокировать, проверить состояние, разблокировать, ждать*", состоит в том, что уведомления, возникающие в интервале времени между операциями *разблокировать и ждать*, могут быть утеряны.

[24] Благодарю Бартоша Милевски за это замечание.

Если функции `wait_for()` и `wait_until()` вызываются без предиката, то они возвращают следующие значения *класса перечисления* (см. раздел 3.1.13):

- `std::cv_status::timeout`, если происходит абсолютная задержка;
- `std::cv_status::no_timeout`, если появилось уведомление.

Если функции `wait_for()` и `wait_until()` вызываются с предикатом, который задается как третий аргумент, то они возвращают результат предиката (индикатор выполнения условия).

Глобальная функция `notify_all_at_thread_exit(cv, l)` вызывает функцию `notify_all()` при выходе из вызывающего потока. Для этого она временно устанавливает соответствующую блокировку *l*, которая должна использовать тот же мьютекс, который используют все ожидающие потоки. Для того чтобы избежать взаимных блокировок, поток должен прекратить выполнение сразу после вызова функции `notify_all_at_thread_exit()`. Таким образом, этот вызов нужен лишь для выполнения очистки перед рассылкой уведомления ожидающим потокам, и эта очистка никогда не должна блокироваться[25].

Класс condition_variable_any

Помимо класса `std::condition_variable`, стандартная библиотека C++ содержит класс `std::condition_variable_any`, не требующий использования объекта класса `std::unique_lock` в качестве блокировки. В документации стандартной библиотеки C++ сказано: "Если вместе с классом `condition_variable_any` используется тип блокировки, отличный от одного из стандартных типов мьютексов или оболочки `unique_lock` для стандартного типа мьютекса, то пользователь должен гарантировать выполнение всех необходимых видов синхронизации по отношению к предикату, связанному с экземпляром класса `condition_variable_any`". Фактически объект должен удовлетворять требованиям концепции *BasicLockable*, требующей предоставления синхронизированных функций-членов `lock()` и `unlock()`.

18.7. Атомарные операции

В первом примере, посвященном условным переменным (см. раздел 18.6.1), мы использовали булеву переменную `readyFlag` в качестве сигнала, который сообщал одному потоку, что другой поток выполнил для него некую операцию или подготовил для него данные. Может возникнуть вопрос: зачем нам по-прежнему нужен мьютекс? Если у нас есть булева переменная, то почему бы нам не позволить одному потоку изменять ее значение, а другому — проверять ее? В момент, когда поток-отправитель изменил значение булевой переменной на `true`, поток-наблюдатель должен иметь возможность увидеть это и выполнить последующую обработку.

Как указано в разделе 18.4, существуют две проблемы.

[25] Типичный пример такой ситуации — сигнализация о конце работы отсоединенного потока (см. раздел 18.2.1). Используя функцию `notify_all_at_thread_exit()`, можно гарантировать, что локальные объекты потока будут уничтожены до того, как главная программа (или главный поток) узнает, что отсоединенный поток прекратил работу.

1. Чтение и запись данных даже элементарного типа не являются атомарными операциями. Таким образом, иногда булево значение оказывается записанным наполовину, что по стандарту приводит к непредсказуемым последствиям.
2. Сгенерированный код может изменить порядок операций, поэтому поток-отправитель может установить флаг готовности до того, как будут подготовлены данные, а поток-получатель может начать обработку данных до установки флага готовности.

С помощью мьютекса можно решить обе проблемы, но мьютекс может оказаться слишком затратной операцией с точки зрения ресурсов и задержки исключительного доступа. Итак, вместо мьютексов и блокировок иногда целесообразно использовать атомарные операции.

В этом разделе впервые вводится *высокоуровневый атомарный интерфейс*, предоставляющий атомарные операции на основе гарантированного порядка доступа к памяти, установленного по умолчанию. Эта гарантия по умолчанию обеспечивает *последовательную консистентность*, означающую, что атомарные операции в потоке выполняются только в запрограммированном порядке. Таким образом, проблемы с переупорядоченными инструкциями, описанные в разделе 18.4.3, не возникают. В конце раздела описывается *низкуровневый атомарный интерфейс*: операции с ослабленными гарантиями порядка.

Отметим, что стандартная библиотека C++ не различает высокоуровневый и низкоуровневый интерфейсы. Термин *низкоуровневый* был введен Гансом Боэмом (Hans Boehm), одним из авторов этой библиотеки. Иногда его также называют *слабым*, или *ослабленным*, атомарным интерфейсом, а высокоуровневый интерфейс иногда называют *обычным*, или *строгим*, атомарным интерфейсом.

18.7.1. Пример использования атомарных операций

Давайте переделаем пример из раздела 18.6.1 в программу с атомарными типами и операциями.

```
#include <atomic> // для атомарных типов
...
std::atomic<bool> readyFlag(false);

void thread1()
{
   // подготовка данных для потока thread2
   ...
   readyFlag.store(true);
}

void thread2()
{
   // ожидаем, пока флаг readyFlag не станет равным true
   // (поток thread1 выполнен)
   while (!readyFlag.load()) {
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
   }

   // выполняем операции, которые должны быть выполнены после того, как
   // поток thread1 закончит подготовку
```

```
   ...
}
```

Сначала мы включили заголовочные файлы <atomic>, в которых объявлены атомарные типы и операции:

```
#include <atomic>
```

Затем объявили атомарный объект, используя шаблонный класс std::atomic<>:

```
std::atomic<bool> readyFlag(false);
```

В принципе в качестве шаблонного параметра можно было бы использовать любой элементарный или целочисленный тип или указатель.

Обратите внимание на то, что атомарные объекты *всегда* следует инициализировать, потому что конструктор по умолчанию не полностью инициализирует их (это не значит, что его начальное значение остается неопределенным, но неинициализированной остается его блокировка)[26]. Статические атомарные объекты необходимо инициализировать константами. Если используется только конструктор по умолчанию, то после него можно выполнить только глобальную операцию atomic_init(), как показано ниже.

```
std::atomic<bool> readyFlag;
...
std::atomic_init(&readyFlag,false);
```

Этот способ инициализации позволяет сохранить совместимость с языком C (см. раздел 18.7.3).

Две наиболее важные операции над атомарными объектами — store() и load().

- Операция store() присваивает новое значение.
- Операция load() возвращает текущее значение.

Важно отметить, что эти операции гарантированно являются атомарными, поэтому теперь не нужен мьютекс для установки флага готовности, который использовался в отсутствие атомарности. Итак, в первом потоке вместо

```
{
   std::lock_guard<std::mutex> lg(readyMutex);
   readyFlag = true;
} // освобождаем блокировку
```

можно просто написать

```
readyFlag.store(true);
```

Во втором потоке вместо

```
{
   std::unique_lock<std::mutex> l(readyFlagMutex);
   while (!readyFlag) {
      l.unlock();
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
```

[26] Благодарю Лоуренса Кроула (Lawrence Crowl) за это замечание.

```
      l.lock();
   }
} // освобождаем блокировку
```

достаточно написать

```
while (!readyFlag.load()) {
   std::this_thread::sleep_for(std::chrono::milliseconds(100));
}
```

Однако для работы с условной переменной по-прежнему нужен мьютекс, который будет получать от нее сообщения.

```
// ждем, пока поток thread1 не будет готов (флаг readyFlag равен true)
{
   std::unique_lock<std::mutex> l(readyMutex);
   readyCondVar.wait(l, []{ return readyFlag.load(); });
} // освобождаем блокировку
```

Для атомарных типов по-прежнему полезны обычные операции, такие как присваивания, автоматические преобразования в целочисленные типы, инкрементация, декрементация и т.д.

```
std::atomic<bool> ab(false);
   ab = true;
   if (ab) {
       ...
}

std::atomic<int> ai(0);
int x = ai;
ai = 10;
ai++;
ai-=17;
```

Однако при работе с атомарными типами эти операции могут выполняться несколько необычно. Например, оператор присваивания возвращает присвоенное значение, а не ссылку на атомарное значение, которое было присвоено. (Подробности изложены в разделе 18.7.2.)

Рассмотрим полный пример с атомарными типами:

```
// concurrency/atomics1.cpp

#include <atomic> // для атомарных типов
#include <future> // для async() и фьючерсов
#include <thread> // для потока this_thread
#include <chrono> // для интервалов времени
#include <iostream>

long data;
std::atomic<bool> readyFlag(false);

void provider ()
{
   // после чтения символа
```

```
   std::cout << "<return>" << std::endl;
   std::cin.get();

   // подготавливаем данные
   data = 42;

   // и сигнал готовности
   readyFlag.store(true);
}

void consumer ()
{
   // ожидаем сигнала готовности и выполняем другие операции
   while (!readyFlag.load()) {
      std::cout.put('.').flush();
      std::this_thread::sleep_for(std::chrono::milliseconds(500));
   }
   // обрабатываем поступившие данные
   std::cout << "\nvalue : " << data << std::endl;
}

int main()
{
   // запускаем потоки provider и consumer
   auto p = std::async(std::launch::async,provider);
   auto c = std::async(std::launch::async,consumer);
}
```

Здесь поток `provider()` сначала поставляет переменную `data`, а затем с помощью функции `store()` сигнализирует о том, что данные готовы:

```
data = 42;              // подготавливаем данные
readyFlag.store(true);  // и сигнал готовности
```

Операция `store()` выполняет так называемую операцию *освобождения* (release) соответствующей ячейки памяти. По умолчанию это гарантирует, что все предыдущие операции с памятью, атомарные или нет, становятся видимыми другим потокам до выполнения операции `store()`.

Соответственно поток `consumer()` выполняет цикл операций `load()`, а затем обрабатывает переменную `data`:

```
while (!readyFlag.load()) {      // цикл до появления сигнала о готовности
...
}
std::cout << data << std::endl;  // обработка полученных данных
```

Операция `load()` выполняет так называемую операцию *захвата* (acquire) ячейки памяти, которая по умолчанию гарантирует, что все следующие операции с памятью, атомарные или нет, становятся видимыми для других потоков после выполнения операции `load()`.

В результате, поскольку запись в переменную `data` *происходит до того*, как поток `provider()` сохраняет значение `true` в переменной `readyFlag`, а обработка переменной `data` происходит после того, как поток `consumer()` загрузит значение `true` как

значение переменной readyFlag, обработка переменной data гарантированно происходит после поступления данных.

Эта гарантия действует, потому что во всех атомарных операциях используется *порядок использования памяти, установленный по умолчанию*, под названием memory_order_seq_cst, который означает *порядок последовательной согласованности*. При работе с низкоуровневыми атомарными операциями существует возможность ослабить эту гарантию (см. раздел 18.7.4).

18.7.2. Подробное описание атомарных типов и их низкоуровневого интерфейса

Основные функциональные возможности атомарных типов описаны в заголовочном файле <atomic> шаблонного класса std::atomic<>. Их можно применять также и ко всем элементарным типам. Предусмотрены специализации для типа bool, всех целочисленных типов и указателей:

```
template<typename T> struct atomic;      // первичный шаблонный класс
template<> struct atomic<bool>;          // явные специализации
template<> struct atomic<int>;
...
template<typename T> struct atomic<T*>; // частичная специализация
                                         // для указателей
```

Высокоуровневые операции над атомарными объектами перечислены в табл. 18.11. По мере возможности они непосредственно переводятся в соответствующие инструкции центрального процессора. Столбец ***triv*** обозначает операции, предусмотренные для типа std::atomic<bool> и других элементарных типов; столбец ***int type*** означает операции для класса std::atomic<>, если используется целочисленный тип, а столбец ***ptr type*** обозначает операции, предусмотренные для класса std::atomic<>, если используется тип указателей.

Сделаем несколько замечаний, касающихся этой таблицы.

- В основном операции возвращают копии, а не ссылки.
- Конструктор по умолчанию не полностью инициализирует переменную/объект. Единственной разрешенной операцией после выполнения конструктора по умолчанию является функция atomic_init(), инициализирующая объект (см. раздел 18.7.1).
- Конструктор для значения соответствующего типа не является атомарным.
- Все функции, за исключением конструкторов, перегружены для типов с модификатором volatile и без него.

Таблица 18.11. Высокоуровневые операции над атомарными типами

Операция	*triv*	*int type*	*ptr type*	Описание
atomic a=*val*	Да	Да	Да	Инициализирует a значением *val* (не атомарная операция)
atomic a; atomic_init(&a,*val*)	Да	Да	Да	То же самое (без atomic_init(), объект a не инициализируется)

Окончание табл. 18.11

Операция	*triv*	*int type*	*ptr type*	Описание
`a.is_lock_free()`	Да	Да	Да	`true`, если тип не использует блокировки внутри
`a.store(`*val*`)`	Да	Да	Да	Присваивает *val* (возвращает `void`)
`a.load()`	Да	Да	Да	Возвращает копию значения `a`
`a.exchange(`*val*`)`	Да	Да	Да	Присваивает *val* и возвращает копию старого значения `a`
`a.compare_exchange_strong(`*exp*, *des*`)`	Да	Да	Да	Операция CAS (сравнение с обменом — compare and swap)
`a.compare_exchange_weak(`*exp*, *des*`)`	Да	Да	Да	Слабая операция CAS
`a =` *val*	Да	Да	Да	Присваивает и возвращает копию *val*
`a.operator` ***atomic***`()`	Да	Да	Да	Возвращает копию значения `a`
`a.fetch_add(`*val*`)`		Да	Да	Атомарная операция `t+=`*val* (возвращает копию нового значения)
`a.fetch_sub(`*val*`)`		Да	Да	Атомарная операция `t-=`*val* (возвращает копию нового значение)
`a +=` *val*		Да	Да	Эквивалентно `t.fetch_add(`*val*`)`
`a -=` *val*		Да	Да	Эквивалентно `t.fetch_sub(`*val*`)`
`++a, a++`		Да	Да	Вызывает `t.fetch_add(1)` и возвращает копию `a` или `a+1`
`--a, a--`		Да	Да	Вызывает `t.fetch_sub(1)` и возвращает копию `a` или `a-1`
`a.fetch_and(`*val*`)`		Да		Атомарная операция `a&=`*val* (возвращает копию нового значения)
`a.fetch_or(`*val*`)`		Да		Атомарная операция `a|=`*val* (возвращает копию нового значения)
`a.fetch_xor(`*val*`)`		Да		Атомарная операция `a^=`*val* (возвращает копию нового значения)
`a &=` *val*		Да		Эквивалентно `a.fetch_and(`*val*`)`
`a |=` *val*		Да		Эквивалентно `a.fetch_or(`*val*`)`
`a ^=` *val*		Да		Эквивалентно `a.fetch_xor(`*val*`)`

Например, для типа `atomic<int>` объявлены следующие операции присваивания:

```
namespace std {
   // специализация std::atomic<> для in:
   template<> struct atomic<int> {
      public:
         // обычные операции присваивания не предусмотрены:
         atomic& operator=(const atomic&) = delete;
         atomic& operator=(const atomic&) volatile = delete;
```

```
        // но предусмотрено присваивание для типа int,
        // возвращающее передаваемый аргумент:
        int operator= (int) volatile noexcept;
        int operator= (int) noexcept;
        ...
    };
}
```

С помощью функции `is_lock_free()` можно проверить, использует ли атомарный тип атомарные блокировки внутренним образом. Если нет, необходима аппаратная поддержка для атомарных операций (это является предварительным условием для использования атомарных типов для обработки сигналов).

Функции `compare_exchange_strong()` и `compare_exchange_weak()` выполняют так называемые операции CAS (*compare-and-swap* — сравнение с обменом). Центральные процессоры часто поддерживают эту атомарную операцию для сравнения содержимого ячейки памяти с заданным значением и, только если они совпадают, модификации содержимого ячейки с помощью данного значения. Это гарантирует, что новое значение вычисляется на основе текущей информации. Такую процедуру можно описать с помощью следующего псевдокода:

```
bool compare_exchange_strong (T& expected, T desired)
{
    if (this->load() == expected) {
        this->store(desired);
        return true;
    }
    else {
        expected = this->load();
        return false;
    }
}
```

Таким образом, если значение было изменено другим потоком, эта функция вернет значение `false` и новое значение в переменной `expected`.

Слабая форма порядка доступа к памяти может привести к внезапному отказу, поэтому функция возвращает значение `false` даже тогда, когда ожидаемое значение существует. Однако слабая форма иногда оказывается эффективнее сильной.

18.7.3. Интерфейс атомарных типов в стиле языка C

Атомарные средства языка C++ имеют аналоги в языке C, которые должны обеспечивать такую же семантику, но, конечно, не могут использовать такие специфические возможности языка C++, как шаблоны, ссылки и функции-члены. Следовательно, весь атомарный интерфейс имеет эквивалент в языке C, который стал частью расширения его стандарта.

Например, можно объявить тип `atomic<bool>` как `atomic_bool`, а вместо функций `store()` и `load()` вызвать глобальные функции, использующие указатель на объект:

```
std::atomic_bool ab;            // эквивалент std::atomic<bool> ab
std::atomic_init(&ab,false);    // см. раздел 18.7.1
...
```

```
std::atomic_store(&ab,true);   // эквивалент ab.store(true)
...
if (std::atomic_load(&ab)) {   // эквивалент if (ab.load())
...
}
```

Однако в язык C добавлен другой интерфейс, использующий суффиксы `_Atomic` и `_Atomic()`, так что интерфейс в стиле языка C в принципе полезен только тогда, когда требуется как можно быстрее написать код, совместимый с языками C и C++.

Тем не менее использование атомарных типов языка C довольно часто встречается в программах на языке C++. В табл. 18.12 перечислены наиболее важные атомарные типы. Кроме них существуют и другие, менее распространенные аналоги, такие как `atomic_int_fast32_t` для класса `atomic<int_fast32_t>`.

Таблица 18.12. Некоторые именованные специализации `std::atomic<>`

Именованный тип в C	Соответствующий тип в C++
`atomic_bool`	`atomic<bool>`
`atomic_char`	`atomic<char>`
`atomic_schar`	`atomic<signed char>`
`atomic_uchar`	`atomic<unsigned char>`
`atomic_short`	`atomic<short>`
`atomic_ushort`	`atomic<unsigned short>`
`atomic_int`	`atomic<int>`
`atomic_uint`	`atomic<unsigned int>`
`atomic_long`	`atomic<long>`
`atomic_ulong`	`atomic<unsigned long>`
`atomic_llong`	`atomic<long long>`
`atomic_ullong`	`atomic<unsigned long long>`
`atomic_char16_t`	`atomic<char16_t>`
`atomic_char32_t`	`atomic<char32_t>`
`atomic_wchar_t`	`atomic<wchar_t>`
`atomic_intptr_t`	`atomic<intptr_t>`
`atomic_uintptr_t`	`atomic<uintptr_t>`
`atomic_size_t`	`atomic<size_t>`
`atomic_ptrdiff_t`	`atomic<ptrdiff_t>`
`atomic_intmax_t`	`atomic<intmax_t>`
`atomic_uintmax_t`	`atomic<uintmax_t>`

Отметим, что для разделяемых указателей (см. раздел 5.2.1) предусмотрены специальные атомарные операции. Причина заключается в том, что такое объявление, как `atomic<shared_ptr<T>>`, невозможно, так как разделяемый указатель нелегко копировать. Атомарные операции подчиняются соглашениям об именовании для интерфейса в стиле языка C (см. раздел 5.2.4).

18.7.4. Низкоуровневый интерфейс атомарных типов

Низкоуровневый интерфейс атомарных типов означает использование атомарных операций без гарантии последовательной согласованности. Таким образом, компиляторы и аппаратное обеспечение могут (частично) изменять порядок доступа к памяти при выполнении атомарных операций (см. раздел 18.4.3).

Напоминаем: несмотря на приведенный пример, эта область напоминает минное поле. Для того чтобы учесть изменение порядка доступа к памяти, необходимо иметь большой опыт, так что даже эксперты часто делают ошибки в таких программах[27].

Эксперт, использующий эту возможность, должен хорошо знать материал, изложенный в работах [*N2480:MemMod*] и [*BoehmAdve:MemMod*], или хотя бы все, что написано в работе [*Boehm:C++MM*].

Пример низкоуровневого интерфейса атомарных типов

Рассмотрим второй пример использования атомарных типов, введенных в разделе 18.7.1, где для управления доступом к переменной был использован атомарный флаг:

```
long data;
std::atomic<bool> readyFlag(false);
```

поток, поставляющий данные:

```
data = 42;                    // задаем данные
readyFlag.store(true);        // устанавливаем сигнал готовности
```

и поток, получающий данные:

```
while (!readyFlag.load()) {     // цикл, пока не появится сигнал готовности
...
}
std::cout << data << std::endl; // обрабатываем полученные данные
```

Поскольку мы используем порядок доступа к памяти, установленный по умолчанию и гарантирующий последовательную согласованность, программа работает так, как написано в разделе 18.7.1. Фактически мы выполняем здесь инструкции

```
data = 42;
readyFlag.store(true,std::memory_order_seq_cst);
```

и

```
while (!readyFlag.load(std::memory_order_seq_cst)) {
   ...
}
std::cout << data << std::endl;
```

Таким образом, каждая операция имеет необязательный аргумент для передачи порядка доступа к памяти, который по умолчанию представлен значением `std::memory_order_seq_cst` (*sequential consistent memory order* — порядок доступа к памяти, обеспечивающий последовательную согласованность).

---

[27] Выражаю особую благодарность Гансу Боэму и Бартошу Милевски за поддержку и помощь при выборе правильного описания данной проблемы. Все возможные ошибки — это моя вина.

Передавая другие значения, можно ослаблять гарантии порядка. В данном случае достаточно потребовать, чтобы поставщик не задерживал выполнение операций после атомарного сохранения и получатель не передвигал вперед операции, следующие за атомарной загрузкой:

```
data = 42;
readyFlag.store(true,std::memory_order_release);
```

и

```
while (!readyFlag.load(std::memory_order_acquire)) {
...
}
std::cout << data << std::endl;
```

Однако ослабление всех ограничений на порядок атомарных операций может привести к непредсказуемым последствиям:

```
// ОШИБКА: неопределенное поведение
data = 42;
readyFlag.store(true,std::memory_order_relaxed);
```

Причина заключается в том, что значение `std::memory_order_relaxed` не гарантирует, что все предыдущие операции с памятью станут видимыми для других потоков до появления результата операции сохранения. Таким образом, поставщик может записать переменную `data` после установки флага готовности, а получатель может прочитать переменную `data` до ее сохранения. Это приводит к *состязанию за данные* (data race).

Отметим, что переменная `data` должна иметь атомарный тип и использовать порядок доступа к памяти `std::memory_order_relaxed`.

```
std::atomic<long> data(0);
std::atomic<bool> readyFlag(false);

// поток-поставщик
data.store(42,std::memory_order_relaxed);
readyFlag.store(true,std::memory_order_relaxed);

// поток-получатель
while (!readyFlag.load(std::memory_order_relaxed)) {
...
}
std::cout << data.load(std::memory_order_relaxed) << std::endl;
```

Строго говоря, это не *неопределенное поведение*, потому что *состязание за данные* не возникает. Однако программа все же работает не так, как ожидается, потому что итоговое значение переменной `data` может еще не быть равным `42` (порядок доступа к памяти не гарантируется). В результате значение переменной `data` может остаться *неопределенным*.

Порядок `memory_order_relaxed` может быть полезным, только если в программе есть атомарные переменные, запись и/или чтение которых выполняются независимо друг от друга. Примером может быть глобальный счетчик, который разные потоки могут увеличивать и уменьшать и в котором нас интересует только его последнее значение, после того как все потоки будут завершены.

Обзор низкоуровневых операций

В табл. 18.13 перечислены вспомогательные низкоуровневые операции для атомарных типов. Как видим, операции загрузки, сохранения, обмена, CAS и извлечения обеспечивают дополнительную возможность задавать порядок доступа к памяти с помощью дополнительного аргумента.

Таблица 18.13. Вспомогательные низкоуровневые операции для атомарных типов

Операция	`triv`	`int type`	`ptr type`
`a.store(`*val*`,`*mo*`)`	Да	Да	Да
`a.load(`*mo*`)`	Да	Да	Да
`a.exchange(`*val*`,`*mo*`)`	Да	Да	Да
`a.compare_exchange_strong(`*exp*`,`*des*`,`*mo*`)`	Да	Да	Да
`a.compare_exchange_strong(`*exp*`,`*des*`,`*mo1*`,`*mo2*`)`	Да	Да	Да
`a.compare_exchange_weak(`*exp*`,`*des*`,`*mo*`)`	Да	Да	Да
`a.compare_exchange_weak(`*exp*`,`*des*`,`*mo1*`,`*mo2*`)`	Да	Да	Да
`a.fetch_add(`*val*`,`*mo*`)`		Да	Да
`a.fetch_sub(`*val*`,`*mo*`)`		Да	Да
`a.fetch_and(`*val*`,`*mo*`)`		Да	
`a.fetch_or(`*val*`,`*mo*`)`		Да	
`a.fetch_xor(`*val*`,`*mo*`)`		Да	

Для ручного управления порядком доступа к памяти существуют дополнительные функции. Например, функции `atomic_thread_fence()` и `atomic_signal_fence()` позволяют вручную поставить барьеры для программы, ограничивающие изменение порядка доступа к памяти.

Без подробностей

Я *не стал объяснять* эти низкоуровневые интерфейсы более подробно, потому что эта информация предназначена для действительно опытных специалистов в области параллельного программирования или тех, кто ими хочет стать. Таким образом, интересующиеся читатели могут обратиться к специальным книгам на эту тему.

Для начала полезно прочитать книгу Anthony Williams *C++ Concurrency in Action* (см. [*Williams:C++Conc*]), особенно главы 5 и 7. Кроме того, Ганс Боэм на своей веб-странице привел адреса веб-страниц, посвященных моделям памяти (см. [*Boehm:C++MM*]).

Глава 19

Распределители памяти

Распределители памяти, введенные в разделе 4.6, представляют собой специальную модель памяти и являются абстракцией, которая используется для преобразования *потребности* в памяти в непосредственное *обращение* к ней. В настоящей главе описываются распределители памяти и соответствующие низкоуровневые функциональные возможности для работы с ней. Подробности изложены в приложении к книге, которое размещено на веб-сайте по адресу `http://www.cppstdlib.com`.

19.1. Использование распределителей памяти с точки зрения прикладного программиста

С точки зрения прикладного программиста использование разных распределителей памяти не должно быть проблемой. Для этого достаточно передать распределитель памяти как шаблонный параметр. Например, следующие операторы создают разные контейнеры и строки с помощью специального распределителя памяти `MyAlloc<>`:

```
// вектор со специальным распределителем памяти
std::vector<int,MyAlloc<int>> v;

// отображение int/float со специальным распределителем памяти
std::map<int,float,std::less<int>,
MyAlloc<std::pair<const int,float>>> m;

// строка со специальным распределителем памяти
std::basic_string<char,std::char_traits<char>,MyAlloc<char>> s;
```

При использовании собственного распределителя памяти, вероятно, целесообразно дать определения нескольких типов. Рассмотрим пример:

```
// специальный строковый тип, использующий специальный распределитель памяти
typedef std::basic_string<char,std::char_traits<char>,
MyAlloc<char>> MyString;

// специальный тип отображения string/string,
// использующий специальный распределитель памяти
typedef std::map<MyString,MyString,std::less<MyString>,
MyAlloc<std::pair<const MyString,MyString>>> MyMap;

// создаем объект указанного типа
MyMap mymap;
```

После принятия стандарта C++11 можно использовать *псевдонимы шаблонов* (alias templates), т.е. объявления шаблонов с помощью операций `typedef` (см. раздел 3.1.9), позволяющие определить тип распределителя памяти без указания типа элементов.

```
template <typename T>
using Vec = std::vector<T,MyAlloc<T>>; // вектор, использующий собственный
                                       // распределитель памяти
Vec<int> coll; // эквивалент std::vector<int,MyAlloc<int>>
```

При работе с объектами, использующими нестандартный распределитель памяти, разница не заметна.

Для того чтобы проверить, не используют ли два распределителя памяти один и тот же ресурс, предусмотрены операции `==` и `!=`. Если оператор `==` возвращает значение `true`, то память, выделенную одним распределителем, можно освободить с помощью другого. Для того чтобы получить доступ к распределителю памяти, все типы, параметризованные распределителем памяти, имеют функцию-член `get_allocator()`. Например:

```
if (mymap.get_allocator() == s.get_allocator()) {
// OK, mymap и s используют один и тот же или эквивалентные
// распределители памяти
...
}
```

Кроме того, после принятия стандарта C++11 появилось свойство (см. раздел 5.4) для проверки наличия у типа `T` шаблонного параметра `allocator_type`, в который можно преобразовать передаваемый распределитель памяти.

```
std::uses_allocator<T,Alloc>::value // true, если тип Alloc допускает
                                    // преобразование в T::allocator_type
```

19.2. Пользовательский распределитель памяти

Распределители памяти имеют интерфейс для выделения и освобождения памяти, а также создания и уничтожения объектов (табл. 19.1). Контейнеры и алгоритмы можно параметризовать распределителями памяти с учетом типа хранящихся в них элементов. Например, можно реализовать распределитель памяти, использующий общую память или отображающий элементы в сохраняемую базу данных.

Таблица 19.1. Основные операции над распределителями памяти

Выражение	Описание
`a.allocate(num)`	Выделяет память для `num` элементов
`a.construct(p,val)`	Инициализирует элемент, на который ссылается указатель `p`, значением `val`
`a.destroy(p)`	Уничтожает элемент, на который ссылается указатель `p`
`a.deallocate(p,num)`	Освобождает память от `num` элементов, на которые ссылается указатель `p`

Написать свой собственный распределитель памяти не очень сложно. Самым важным аспектом является способ, с помощью которого выделяется или освобождается память. После принятия стандарта C++11 остальные факторы, как правило, можно задать по умолчанию. (До принятия стандарта C++11 их необходимо было реализовать явным образом.) Как пример, рассмотрим распределитель памяти, работающий аналогично распределителю памяти, заданному по умолчанию:

```
// alloc/myalloc11.hpp

#include <cstddef>        // для типа size_t

template <typename T>
class MyAlloc {
   public:
     // определения типов
     typedef T value_type;

     // конструкторы
     // - ничего не делают, потому что распределитель памяти не имеет состояния
     MyAlloc () noexcept {
     }
     template <typename U>
     MyAlloc (const MyAlloc<U>&) noexcept {
         // нет состояния для копирования
     }

     // выделяем память для num элементов типа T, но не инициализируем ее
     T* allocate (std::size_t num) {
         // выделяем память с помощью глобальной операции new
         return static_cast<T*>(::operator new(num*sizeof(T)));
     }

     // освобождаем память, на которую ссылается указатель p,
     // от удаленных элементов
     void deallocate (T* p, std::size_t num) {

     // освобождаем память с помощью глобальной операции delete
     ::operator delete(p);
   }
};

// возвращаем признак того, что все специализации данного
// распределителя памяти являются эквивалентными
template <typename T1, typename T2>
bool operator== (const MyAlloc<T1>&,
                 const MyAlloc<T2>&) noexcept {
   return true;
}
template <typename T1, typename T2>
bool operator!= (const MyAlloc<T1>&,
                 const MyAlloc<T2>&) noexcept {
   return false;
}
```

Как следует из этого примера, программист должен реализовать следующие функциональные возможности.

- Определение типа `value_type`, представляющего собой всего лишь тип передаваемого шаблонного параметра.
- Конструктор.
- Шаблонный конструктор, копирующий внутреннее состояние при изменении типа. Отметим, что шаблонный конструктор не подавляет неявное объявление конструктора копирования (см. раздел 3.2).
- Член `allocate()`, выделяющий новую память.
- Член `deallocate()`, освобождающий память, которая больше не нужна.
- Конструктор и деструктор (при необходимости) для инициализации, копирования и очистки внутреннего состояния.
- Операции `==` и `!=`.

Функции `construct()` или `destroy()` реализовывать не обязательно, потому что их реализации по умолчанию обычно работают отлично (используя операцию *new с размещением* для инициализации памяти и вызывая деструктор для ее явной очистки).

Используя эту базовую реализацию, легко создать свой собственный распределитель памяти. Для реализации своей стратегии выделения памяти можно использовать функции `allocate()` и `deallocate()`. Эта стратегия может подразумевать повторное использование памяти вместо ее немедленного освобождения, использование общей памяти, отображение памяти в сегмент объектно-ориентированной базы данных или просто отладку распределителя памяти. Кроме того, программист может предусмотреть свои конструкторы и деструктор для выделения и освобождения памяти вместо функций `allocate()` и `deallocate()`. До принятия стандарта C++11 класс распределителя памяти должен был содержать намного больше функций-членов, которые, впрочем, легко можно было написать. Законченный пример, демонстрирующий распределители памяти, можно найти в файле `alloc/myalloc03.hpp`, который описан также в специальном разделе, посвященном распределителям памяти, на веб-сайте `http://www.cppstdlib.com`.

19.3. Использование распределителей памяти с точки зрения разработчика библиотеки

В этом разделе описывается использование распределителей памяти с точки зрения разработчиков контейнеров и других компонентов, способных работать с разными распределителями памяти. Этот раздел частично (с разрешения автора) содержит материал из раздела 19.4 книги Бьярне Страуструпа (Bjarne Stroustrup) *The C++ Programming Language*, 3rd edition (см. [*Stroustrup:C++*]).

В качестве примера рассмотрим наивную реализацию вектора. Вектор получает свой распределитель памяти как шаблонный параметр или аргумент конструктора и сохраняет его в какой-то области памяти.

```
namespace std {
   template <typename T,
```

```
            typename Allocator = allocator<T> >
   class vector {
      ...
   private:
      Allocator alloc;      // распределитель
      T* elems;             // массив элементов
      size_type numElems;   // количество элементов
      size_type sizeElems; // размер памяти для элементов
      ...
   public:
      // конструкторы
     explicit vector(const Allocator& = Allocator());
     explicit vector(size_type num, const T& val = T(),
                     const Allocator& = Allocator());
     template <typename InputIterator>
     vector(InputIterator beg, InputIterator end,
            const Allocator& = Allocator());
     vector(const vector<T,Allocator>& v);
     ...
   };
}
```

Отметим, что, строго говоря, тип элементов по стандарту C++11 должен быть следующим:

```
allocator_traits<Allocator>::pointer elems;
```

Как и свойства итератора (см. раздел 9.5), *свойства распределителя памяти* были изобретены для того, чтобы они служили общим интерфейсом для обобщенного кода, работающего с распределителями памяти. Они поддерживают такие типы, как `pointer`, и такие операции, как `allocate()`, `construct()`, `destroy()` и `deallocate()`.

Второй конструктор, инициализирующий вектор `num` элементами со значением `val`, должен быть реализован следующим образом:

```
namespace std {
   template <typename T, typename Allocator>
   vector<T,Allocator>::vector(size_type num, const T& val,
                               const Allocator& a)
   : alloc(a) // инициализируем распределитель
   {
      // выделяем память
      sizeElems = numElems = num;
      elems = allocator_traits<Allocator>::allocate(alloc, num);

      // инициализируем элементы
      for (size_type i=0; i<num; ++i) {

          // инициализируем i-й элемент
          allocator_traits<Allocator>::construct(alloc, &elems[i], val);
       }
    }
}
```

Отметим, что этот код не полный, потому что в нем нет обработки исключений. В правильной реализации после неудачного создания любого элемента вся выделенная память должна быть освобождена.

Таблица 19.2. Вспомогательные функции для работы с неинициализированной памятью

Выражение	Описание
`uninitialized_fill(`*beg*`,`*end*`,`*val*`)`	Инициализирует интервал [*beg*,*end*) значением *val*
`uninitialized_fill_n(`*beg*`,`*num*`,`*val*`)`	Инициализирует *num* элементов, начиная с позиции *beg*, значением *val*
`uninitialized_copy(`*beg*`,`*end*`,`*mem*`)`	Инициализирует элементы, начиная с позиции *mem*, элементами интервала [*beg*,*end*)
`uninitialized_copy_n(`*beg*`,`*num*`,`*mem*`)`	Инициализирует *num* элементов, начиная с позиции *mem*, элементами, начиная с позиции *beg* (по стандарту C++11)

Однако для инициализации неинициализированной памяти стандартная библиотека C++ содержит несколько вспомогательных функций (табл. 19.2). С помощью этих функций реализация конструктора становится еще проще:

```
namespace std {
   template <typename T, typename Allocator>
   vector<T,Allocator>::vector(size_type num, const T& val,
                               const Allocator& a)
   : alloc(a) // инициализируем распределитель
   {

      // выделяем память
      sizeElems = numElems = num;
      elems = alloc.allocate(num);

      // инициализируем элементы
      uninitialized_fill_n(elems, num, val);
   }
}
```

Однако по стандарту C++11 функции `uninitialized_fill_n()` и `uninitialized_copy()` применять нельзя, потому что при создании элементов они не используют свойства распределителя памяти. В этом случае управление памятью может быть нарушено пользовательским кодом, в котором определены свойства распределителя памяти и/или типы распределителей памяти, в которых вызов функции `construct()` может приводить в выполнению дополнительных или совершенно других операций.

Функцию-член `reserve()`, резервирующую дополнительную память без изменения количества элементов (см. раздел 7.3.1), можно реализовать следующим образом:

```
namespace std {
   template <typename T, typename Allocator>
   void vector<T,Allocator>::reserve(size_type size)
```

```
    {
        // функция reserve() никогда не уменьшает размер памяти
        if (size <= sizeElems) {
            return;
        }

        // выделяем новую память для size элементов
        T* newmem = allocator_traits<Allocator>::allocate(alloc, size);

        // копируем старые элементы в новую память
        ...

        // уничтожаем старые элементы
        for (size_type i=0; i<numElems; ++i) {
             allocator_traits<Allocator>::destroy(alloc, &elems[i]);;
        }

        // освобождаем старую память
        allocator_traits<Allocator>::deallocate(alloc,elems,sizeElems);

        // теперь наши элементы записаны в новой памяти
        sizeElems = size;
        elems = newmem;
    }
}
```

Отметим еще раз, что этот код является слишком упрощенным. В нем отсутствует сложная часть, относящаяся к копированию элементов в новую память, потому что при этом используются исключения и по возможности должны выполняться перемещающие, а не копирующие операторы.

Простые итераторы

Для обхода и инициализации неинициализированной памяти предназначен класс `raw_storage_iterator`. Написав алгоритм, использующий класс `raw_storage_iterator`, можно инициализировать память значениями, которые являются результатами этого алгоритма.

Например, следующий оператор инициализирует память, на которую ссылается указатель `elems`, значениями из интервала [`x.begin()`,`x.end()`):

```
copy (x.begin(), x.end(),                    // источник
      raw_storage_iterator<T*,T>(elems));    // получатель
```

Первый шаблонный параметр (в данном случае `T*`) должен быть итератором вывода для указанного типа элементов. Второй шаблонный параметр (в данном случае `T`) должен быть типом элементов.

Временные буфера

В программах иногда встречаются функции `get_temporary_buffer()` и `return_temporary_buffer()`, предназначенные для обработки неинициализированной памяти,

которая используется в этих функциях как кратковременное и временное хранилище. Функция get_temporary_buffer() может вернуть меньше памяти, чем ожидалось. Таким образом, функция get_temporary_buffer() возвращает пару, содержащую адрес памяти и ее размер (количество элементов). Рассмотрим пример использования этих функций:

```
void f()
{
   // выделяем память для num элементов типа MyType
   pair<MyType*,std::ptrdiff_t> p
      = get_temporary_buffer<MyType>(num);
   if (p.second == 0) {
      // невозможно выделить память для элементов
      ...
   }
   else if (p.second < num) {
      // невозможно выделить память для num элементов
      // однако ее следует не забыть освободить
      ...
   }

   // обработка
   ...

   // освобождаем временно выделенную память, если она есть
   if (p.first != 0) {
       return_temporary_buffer(p.first);
   }
}
```

Но с помощью функций get_temporary_buffer() и return_temporary_buffer() довольно сложно написать безопасный с точки зрения исключений код, поэтому они, как правило, не используются в реализациях библиотеки.

Приложение

S.1. Битовые множества

Как было сказано в разделе 12.5, битовые множества имитируют массивы битов булевых значений фиксированного размера и полезны для управления флагами. В старых программах на языках C и C++ для манипулирования массивами битов с помощью побитовых операций, таких как `&`, `|`, и `~`, использовался тип `long`. Преимущество класса `bitset` заключается в том, что битовые множества могут содержать произвольное количество битов и допускают применение дополнительных операций. Например, можно присвоить один бит, а также читать и записывать битовые множества в виде последовательности нулей и единиц.

Изменять количество битов в битовом множестве нельзя, так как оно является шаблонным параметром. Если требуется контейнер для переменного количества битов или булевых значений, можно использовать класс `vector<bool>` (см. раздел 7.3.6).

Класс `bitset` определен в заголовочном файле `<bitset>`.

```
#include <bitset>
```

В заголовочном файле `<bitset>` класс `bitset` определен как шаблонный класс, у которого количество битов является шаблонным параметром.

```
namespace std {
   template <size_t Bits>
   class bitset;
}
```

В данном случае шаблонный параметр является не типом, а целочисленным значением без знака (см. раздел 3.2).

Шаблоны с разными шаблонными параметрами представляют собой разные типы. Сравнивать и комбинировать можно только битовые множества с одинаковым количеством битов.

Новшества стандарта C++11

В стандарте C++98 были перечислены практически все свойства битовых множеств. Ниже приводится список наиболее важных свойств, добавленных в стандарте C++11.

- Битовые множества можно инициализировать строковыми литералами (см. раздел 12.5.1).
- Преобразования битовых множеств в числовые значения, и наоборот, поддерживают тип `unsigned long long`. Для этого была введена функция `to_ullong()` (см. раздел 12.5.1).
- Преобразования битовых множеств в строки, и наоборот, позволяют указывать символ, интерпретируемый как установленный или сброшенный бит.

- Функция-член `all()` позволяет проверять, установлены ли биты.
- Для использования битовых множеств в неупорядоченных контейнерах предусмотрена хеш-функция по умолчанию (см. раздел 7.9.2).

S.1.1. Примеры использования битовых множеств

Использование битовых множеств как наборов флагов

Первый пример демонстрирует использование битовых множеств для манипулирования наборами флагов. Каждый флаг имеет значение, определенное как тип перечисления. Значение типа перечисления используется как позиция бита в битовом множестве. В частности, биты представляют цвета. Таким образом, каждое значение перечисления определяет один цвет. Используя битовые множества, можно управлять переменными, содержащими любое сочетание цветов.

```
// contadapt/bitset1.cpp

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

int main()
{
   // тип перечисления для битов
   // - каждый бит представляет цвет
   enum Color { red, yellow, green, blue, white, black, ...,
                numColors };

   // создаем битовое множество для всех битов/цветов
   bitset<numColors> usedColors;

   // задаем биты для двух цветов
   usedColors.set(red);
   usedColors.set(blue);

   // выводим данные о битовом множестве
   cout << "bitfield of used colors:   " << usedColors << endl;
   cout << "number of used colors:     " << usedColors.count() << endl;
   cout << "bitfield of unused colors: " << ~usedColors << endl;

   // если используется любой цвет
   if (usedColors.any()) {
       // цикл по всем цветам
       for (int c = 0; c < numColors; ++c) {
            // если используется текущий цвет
            if (usedColors[(Color)c]) {
            ...
            }
       }
   }
}
```

Использование битовых множеств для ввода и вывода бинарных представлений

Полезным свойством битовых множеств является способность преобразовывать целочисленные значения в последовательность битов, и наоборот. Эту операцию легко выполнить с помощью временного битового множества.

```
// contadapt/bitset2.cpp

#include <bitset>
#include <iostream>
#include <string>
#include <limits>
using namespace std;

int main()
{
   // выводим несколько чисел в бинарном представлении
   cout << "267 as binary short:      "
        << bitset<numeric_limits<unsigned short>::digits>(267)
        << endl;
   cout << "267 as binary long:       "
        << bitset<numeric_limits<unsigned long>::digits>(267)
        << endl;
   cout << "10 000 000 with 24 bits:  "
        << bitset<24>(1e7) << endl;

   // записываем бинарное представление в строку
   string s = bitset<42>(12345678).to_string();
   cout << "12 345 678 with 42 bits:  " << s << endl;

   // преобразуем бинарное представление в целочисленное значение
   cout << "\"1000101011\" as number:   "
        << bitset<100>("1000101011").to_ullong() << endl;
}
```

В зависимости от количества битов, используемых для представления типов `short` и `long long`, программа может выводить следующий результат:

```
267 as binary short:      0000000100001011
267 as binary long:       00000000000000000000000100001011
10 000 000 with 24 bits:  100110001001011010000000
12 345 678 with 42 bits:  000000000000000000101111000110000101001110
"1000101011" as number:   555
```

В этом примере следующее выражение преобразовывает целое число `267` в битовое множество с количеством битов, которое используется для представления типа `unsigned short` (см. раздел 5.3).

```
bitset<numeric_limits<unsigned short>::digits>(267)
```

Оператор вывода для типа `bitset` выводит биты как последовательность символов `0` и `1`. Кроме того, битовое множество можно вывести сразу или использовать его как строку.

```
string s = bitset<42>(12345678).to_string();
```

Отметим, что до принятия стандарта C++11 для этого необходимо было писать

```
string s = bitset<42>(12345678).to_string<char,char_traits<char>,
                                          allocator<char> >();
```

потому что функция `to_string()` является членом шаблонного класса, а в стандарте не были определены значения по умолчанию для шаблонных параметров.

Аналогично следующее выражение преобразовывает последовательность бинарных символов в битовое множество, которое функция `to_ullong()` превращает в целочисленное значение:

```
bitset<100>("1000101011")
```

Отметим, что количество битов в битовом множестве должно быть меньше `sizeof(unsigned long long)`. Причина заключается в том, что если значение невозможно представить типом `unsigned long long`, то генерируется исключение[1].

Кроме того, до принятия стандарта C++11 необходимо было явно преобразовать начальное значение в тип `string`.

```
bitset<100>(string("1000101011"))
```

S.1.2. Подробное описание класса `bitset<>`

Класс `bitset` предусматривает следующие операции.

Создание, копирование и уничтожение

Для битовых множеств определено несколько конструкторов. Однако для них не предусмотрены специальные копирующие конструкторы, операции присваивания и деструктора. Таким образом, битовые множества присваиваются и копируются с помощью стандартной операции побитового копирования.

`bitset`<*bits*>**::bitset** ()

- Конструктор по умолчанию.
- Создает битовое множество, в котором все биты инициализированы нулем.
- Например:

```
bitset<50> flags; // флаги: 0000...000000
// таким образом, имеем 50 сброшенных битов
bitset<bits>::bitset (unsigned long long value)
```

`bitset`<*bits*>**::bitset** (`unsigned long long` *value*)

- Создает битовое множество, инициализированное битами целочисленного значения *value*.

[1] До принятия стандарта C++11 тип `unsigned long` не поддерживался, поэтому в этом месте можно было вызвать только функцию `to_ulong()`. Функцию `to_ulong()` все еще можно вызывать, если количество битов меньше, чем `sizeof(unsigned long)`.

- Если количество битов в значении *value* слишком мало, старшие биты инициализируются нулем.
- До принятия стандарта C++11 значение *value* имело тип `unsigned long`.
- Например:

```
bitset<50> flags(7); // флаги 0000...000111
explicit bitset<bits>::bitset (const string& str)
bitset<bits>::bitset (const string& str, string::size_type str_idx)
bitset<bits>::bitset (const string& str,
                      string::size_type str_idx,
                      string::size_type str_num)
bitset<bits>::bitset (const string& str,
                      string::size_type str_idx,
                      string::size_type str_num,
                      string::charT zero)
bitset<bits>::bitset (const string& str,
                      string::size_type str_idx,
                      string::size_type str_num,
                      string::charT zero, string::charT one)
```

- Все версии создают битовое множество, инициализированное строкой *str* или подстрокой строки *str*.
- Строка или подстрока может содержать только символы `'0'` и `'1'` (или *zero* и *one*).
- *str_idx* — это индекс первого символа строки *str*, используемый для инициализации.
- Если параметр *str_num* пропущен, то используются все символы от *str_idx* до конца строки *str*.
- Если строка или подстрока имеет меньше символов, чем требуется, то старшие биты инициализируются нулями.
- Если строка или подстрока имеет больше символов, чем требуется, то остальные символы игнорируются.
- Если *str_idx* > *str*.`size()`, то генерируется исключение `out_of_range`.
- Если один из символов не равен ни `'0'`/*zero*, ни `'1'`/*one*, то генерируется исключение `invalid_argument`.
- Параметры *zero* и *one* появились в стандарте C++11.
- Отметим, что этот конструктор является шаблонным членом класса (см. раздел 3.2). По этой причине не предусмотрено неявное преобразование типа `const char*` в `string` для первого параметра, который до принятия стандарта C++11 должен был быть строковым литералом.
- Например:

```
bitset<50> flags(string("1010101"));     // флаги: 0000...0001010101
bitset<50> flags(string("1111000"),2,3); // флаги: 0000...0000000110

explicit bitset<bits>::bitset (const charT* str)
bitset<bits>::bitset (const charT* str, string::size_type str_num)
bitset<bits>::bitset (const charT* str, string::size_type str_num,
                      string::charT zero)
```

```
bitset<bits>::bitset (const charT* str, string::size_type str_num,
                      string::charT zero, string::charT one)
```

- Все версии создают битовое множество, инициализированное строкой символов *str*.
- Строка или подстрока может содержать символы `'0'` и `'1'` (или *zero* и *one*).
- Если параметр *str_num* пропущен, то используются все символы строки *str*.
- Если строка *str* содержит меньше символов, чем требуется (параметр *str_num* выходит за пределы допустимого диапазона), то старшие биты инициализируются нулем.
- Если параметр *str* содержит больше параметров, чем требуется, то остальные символы игнорируются.
- Если один из символов не равен ни `'0'`/*zero*, ни `'1'`/*one*, то генерируется исключение `invalid_argument`.
- Параметры *zero* и *one* появились в стандарте C++11.
- Например:

```
bitset<50> flags("1010101"); // флаги: 0000...0001010101
```

Немодифицирующие операции

```
size_t bitset<bits>::size () const
```

- Возвращает количество битов (т.е. значение *bits*).

```
size_t bitset<bits>::count () const
```

- Возвращает количество установленных битов (т.е. битов со значением `1`).

```
bool bitset<bits>::all () const
```

- Возвращает признак того, что все биты установлены.
- Начиная со стандарта C++11.

```
bool bitset<bits>::any () const
```

- Возвращает признак того, что хотя бы один бит установлен.

```
bool bitset<bits>::none () const
```

- Возвращает признак того, что ни один бит не установлен.

```
bool bitset<bits>::test (size_t idx) const
```

- Возвращает признак того, что бит в позиции *idx* установлен.
- Если *idx* >= `size()`, генерирует исключение `out_of_range`.

```
bool bitset<bits>::operator == (const bitset<bits>& bits) const
```

- Возвращает признак того, что все биты объекта `*this` и параметра *bits* имеют одно и то же значение.

```
bool bitset<bits>::operator != (const bitset<bits>& bits) const
```

- Возвращает признак того, что все биты объекта `*this` и параметра *bits* имеют разные значения.

Манипулирующие операции

bitset<*bits*>`&` **bitset**<*bits*>**::set** `()`

- Устанавливает все биты равными `true`.
- Возвращает модифицированное битовое множество.

bitset<*bits*>`&` **bitset**<*bits*>**::set** `(size_t` *idx*`)`

- Устанавливает бит в позиции *idx* равным `true`.
- Возвращает модифицированное битовое множество.
- Если *idx* `>= size()`, генерирует исключение `out_of_range`.

bitset<*bits*>`&` **bitset**<*bits*>**::set** `(size_t` *idx*`, bool` *value*`)`

- Устанавливает бит в позиции *idx* в соответствии со значением *value*.
- Возвращает модифицированное битовое множество.
- Если *idx* `>= size()`, генерирует исключение `out_of_range`.
- До принятия стандарта C++11 параметр *value* имел тип `int`, так что значение `0` устанавливало бит равным `false`, а любое другое значение соответствовало значению `true`.

bitset<*bits*>`&` **bitset**<*bits*>**::reset** `()`

- Сбрасывает все биты в значение `false` (присваивает `0` всем битам).
- Возвращает модифицированное битовое множество.

bitset<*bits*>`&` **bitset**<*bits*>**::reset** `(size_t` *idx*`)`

- Сбрасывает бит в позиции *idx* в значение `false`.
- Возвращает модифицированное битовое множество.
- Если *idx* `>= size()`, генерирует исключение `out_of_range`.

bitset<*bits*>`&` **bitset**<*bits*>**::flip** `()`

- Переключает все биты (устанавливает сброшенные биты, и наоборот).
- Возвращает модифицированное битовое множество.

bitset<*bits*>`&` **bitset**<*bits*>**::flip** `(size_t` *idx*`)`

- Переключает бит в позиции *idx*.
- Возвращает модифицированное битовое множество.
- Если *idx* `>= size()`, генерирует исключение `out_of_range`.

`bitset<`*bits*`>& bitset<`*bits*`>::`**operator** `^= (const bitset<`*bits*`>&` *bits*`)`

- Побитовая операция исключающего ИЛИ.

- Переключает все биты, установленные в параметре *bits*, оставляя все остальные биты неизменными.
- Возвращает модифицированное битовое множество.

bitset<*bits*>& **bitset**<*bits*>::**operator** |= (const bitset<*bits*>& *bits*)

- Побитовая операция ИЛИ.
- Устанавливает все биты, установленные в параметре *bits*, оставляя все остальные биты неизменными.
- Возвращает модифицированное битовое множество.

bitset<*bits*>& **bitset**<*bits*>::**operator** &= (const bitset<*bits*>& *bits*)

- Побитовая операция И.
- Сбрасывает все биты, не установленные в параметре *bits*, оставляя все остальные биты неизменными.
- Возвращает модифицированное битовое множество.

bitset<*bits*>& **bitset**<*bits*>::**operator** <<= (size_t *num*)

- Сдвигает все биты влево на *num* позиций.
- Возвращает модифицированное битовое множество.
- Самый младший из *num* битов устанавливается равным `false`.

bitset<*bits*>& bitset<*bits*>::**operator** >>= (size_t *num*)

- Сдвигает все биты вправо на *num* позиций.
- Возвращает модифицированное битовое множество.
- Самый старший из *num* битов устанавливастся равным `false`.

Доступ с помощью операции [ ]

bitset<*bits*>::reference **bitset**<*bits*>::**operator**[ ] (size_t *idx*)
bool **bitset**<*bits*>::**operator**[ ] (size_t *idx*) const

- Обе версии возвращают бит в позиции *idx*.
- Первая версия для неконстантных битов использует тип прокси, чтобы использовать возвращаемое значение как модифицирумое (lvalue). Подробнее см. ниже.
- Вызывающая сторона должна гарантировать, что параметр *idx* является действительным индексом; в противном случае поведение непредсказуемо.

Если операция [ ] применяется к неконстантным битовым множествам, то она возвращает специальный временный объект типа bitset<>::reference. Этот объект используется как прокси[2], допускающий некоторые модификации с битами, доступ к которым

[2] Прокси позволяет сохранять контроль в тех местах, где он обычно не предусмотрен. Он часто используется для повышения безопасности. В данном случае прокси поддерживает контроль, чтобы позволить выполнение некоторых операций, хотя возвращаемое значение в принципе ведет себя как объект типа bool.

обеспечивается операцией [ ]. В частности, для объектов reference предусмотрены пять операций.

1. `reference&` **operator**= `(bool)`

   Устанавливает бит в соответствии с передаваемым значением.

2. `reference&` **operator**= `(const reference&)`

   Устанавливает бит в соответствии с другой ссылкой.

3. `reference&` **flip** `()`

   Переключает значение бита.

4. **operator bool** `() const`

   Автоматически преобразовывает значение в булево.

5. `bool` **operator**~ `() const`

   Возвращает дополнение бита (его переключенное значение).

Например, можно написать следующие операторы:

```
bitset<50> flags;
...
flags[42] = true;              // устанавливем бит 42
flags[13] = flags[42];         // присваиваем значение бита 42 биту 13
flags[42].flip();              // переключаем значение бита 42
if (flags[13]) {               // если бит 13 установлен,
   flags[10] = ~flags[42];     // то присваиваем дополнение бита 42 биту 10
}
```

Создание новых модифицированных битовых множеств

bitset<*bits*> **bitset**<*bits*>::**operator** ~ `() const`

- Возвращает новое битовое множество, в котором все биты имеют значения, обратные значениям в объекте `*this`.

bitset<*bits*> bitset<*bits*>::**operator** << `(size_t` *num*`) const`

- Возвращает новое битовое множество, в котором все биты сдвинуты влево на *num* позиций.

bitset<*bits*> bitset<*bits*>::**operator** >> `(size_t` *num*`) const`

- Возвращает новое битовое множество, в котором все биты сдвинуты вправо на *num* позиций.

bitset<*bits*> **operator** & `(const bitset<`*bits*`>&` *bits1*`,`
`const bitset<`*bits*`>&` *bits2*`)`

- Возвращает новое битовое множество, вычисленное с помощью операции И, примененной к объектам *bits1* и *bits2*.
- Возвращает новое битовое множество, в котором установлены только те биты, которые установлены как в объекте *bits1*, так и в объекте *bits2*.

bitset<*bits*> **operator** | (const bitset<*bits*>& *bits1*,
const bitset<*bits*>& *bits2*)

- Возвращает новое битовое множество, вычисленное с помощью операции ИЛИ, примененной к объектам *bits1* и *bits2*.
- Возвращает новое битовое множество, в котором установлены только те биты, которые установлены или в объекте *bits1*, или в объекте *bits2* (или в обоих).

bitset<*bits*> **operator** ^ (const bitset<*bits*>& *bits1*,
const bitset<*bits*>& *bits2*)

- Возвращает новое битовое множество, вычисленное с помощью операции исключающего ИЛИ, примененной к объектам *bits1* и *bits2*.
- Возвращает новое битовое множество, содержащее только биты, установленные в объекте *bits1*, но не в объекте *bits2*, и наоборот.

Операции преобразования типов

unsigned long **bitset**<*bits*>::**to_ulong** () const

- Возвращает целочисленное значение, которое представляют биты битового множества.
- Если целочисленное значение невозможно представить типом unsigned long, генерирует исключение overflow_error.

unsigned long long **bitset**<*bits*>::**to_ullong** () const

- Возвращает целочисленное значение, которое представляют биты битового множества.
- Если целочисленное значение невозможно представить типом unsigned long long, генерирует исключение overflow_error.
- Начиная со стандарта C++11.

string bitset<*bits*>::**to_string** () const
string bitset<*bits*>::**to_string** (*charT zero*) const
string bitset<*bits*>::**to_string** (*charT zero*, *charT one*) const

- Возвращает строку, содержащую значение битового множества в двоичном виде, в котором символ '0' означает сброшенные биты и '1' — установленные.
- Порядок символов эквивалентен порядку битов с убывающими индексами.
- Параметры *zero* и *one* введены в стандарте C++11.
- Например:

```
bitset<50> b;
std::string s;
...
s = b.to_string();
```

- Это шаблонная функция, параметризованная только типом возвращаемого значения, у которой до принятия стандарта C++11 не было значения, заданного по умолчанию. Таким образом, по старым правилам языка C++11 следовало написать:

```
s = b.to_string<char,char_traits<char>,allocator<char> >()
```

Операции ввода-вывода

istream& **operator** >> (*istream*& *strm*, bitset<*bits*>& *bits*)

- Считывает в объект *bits* битовое множество, представляющее собой последовательность символов '0' и '1'.
- Считывание выполняется до тех пор, пока не произойдет одно из следующих событий.
  - Считано *bits* символов.
  - В потоке *strm* обнаружен конец файла.
  - Следующий символ не совпадает ни с '0', ни с '1'.
- Возвращает *strm*.
- Если количество считанных битов меньше, чем количество битов в битовом множестве, то битовое множество заполняется ведущими нулями.
- Если невозможно считать ни один символ, то эта операция устанавливает в потоке *strm* флаг ios::failbit, который может активизировать соответствующее исключение (см. раздел 15.4.4).

ostream& **operator** << (*ostream*& *strm*, const bitset<*bits*>& *bits*)

- Записывает битовое множество *bits*, преобразованное в строку, содержащую двоичное представление (т.е. представляющую собой последовательность '0' и '1').
- Использует функцию to_string() (см. раздел S.1.2) для создания символов вывода.
- Возвращает поток *strm*.
- См. пример в разделе 12.5.1.

Поддержка хеширования

После принятия стандарта C++11 класс bitset<> имеет специализацию хеш-функции:

```
namespace std {
   template <size_t N> struct hash<bitset<N> >;
}
```

Подробнее см. в разделе 7.9.2.

S.2. Массивы значений

Начиная со стандарта C++98, стандартная библиотека C++ содержит класс valarray<> для обработки массивов, содержащих числовые значения. Массив значений — это представление математической концепции линейной последовательности значений.

Он имеет одну размерность, но создает иллюзию высокой размерности за счет специального способа вычисления индексов и мощные возможности выделения подмножеств. Следовательно, массив значений можно использовать и для векторных и матричных операций, и для эффективной обработки систем полиномиальных уравнений.

Классы массивов значений позволяют осуществлять сложную оптимизацию. Однако неясно, насколько важным окажется этот компонент стандартной библиотеки шаблонов

в будущем, потому что существуют более эффективные и интересные разработки. Одним из наиболее интересных примеров является система Blitz. Если вас интересуют вопросы числовой обработки, обратитесь к системе Blitz. Подробности изложены на веб-сайте по адресу `http://www.oonumerics.org/blitz/`.[3]

Классы массивов значений разработаны не очень хорошо. Фактически никто не может гарантировать, что их окончательная спецификация является работоспособной. Это произошло потому, что никто не чувствовал ответственности за эти классы. Люди, включившие массивы значений в стандартную библиотеку C++, вышли из комитета по стандартизации задолго до окончания работы над стандартом. Например, для того чтобы использовать массивы значений, часто приходится выполнять неудобные и долгие преобразования типов (см. раздел S.2.2).

Новшества стандарта C++11

Стандарт C++98 определял практически все функциональные возможности массивов значений. Перечислим наиболее важные свойства, добавленные в стандарте C++11.

- Массивы значений поддерживают семантику перемещения. Для них существуют перемещающий конструктор и перемещающая операция присваивания.
- Класс содержит функции `swap()` (см. раздел S.2.3).
- Можно использовать список инициализации для заполнения массива значений начальными значениями или присваивания ему новых значений (см. раздел S.2.1).
- Класс массивов значений имеет функции `begin()` и `end()` для обхода своих элементов (см. раздел S.2.1).
- Операция `[ ]` возвращает константную ссылку, а не копию (см. раздел S.2.3).

S.2.1. Описание массивов значений

Массивы значений — это одномерные массивы с элементами, последовательно пронумерованными начиная с нуля. Они реализуют возможность выполнять определенные числовые процедуры над всеми или частью значений в одном или нескольких массивах.

Например, можно выполнить оператор

```
z = a*x*x + b*x + c
```

в котором переменные `a`, `b`, `c`, `x` и `z` представляют собой массивы, содержащие сотни числовых значений, что позволяет упростить программу. Кроме того, эти операции над массивами значений выполняются эффективно, потому что классы выполняют определенную оптимизацию, чтобы избежать создания временных объектов при вычислении оператора. Кроме того, специальные интерфейсы и вспомогательные классы предоставляют возможность обрабатывать только одно определенное подмножество массивов значений или имитировать работу с многомерными массивами. Таким образом, массивы значений представляют собой концепцию, помогающую реализовать векторные и матричные операции, а также реализующие их классы.

[3] Актуальность ссылок не гарантируется. — *Примеч. ред.*

Стандарт гарантирует, что массивы значений не имеют псевдонимов. Иначе говоря, к любому значению неконстантного массива значений можно обратиться по уникальному пути. Таким образом, операции над этими значениями можно оптимизировать, потому что компилятор не учитывает данные, к которым обращаются по другому пути.

Заголовочный файл

Массивы значений объявлены в заголовочном файле <valarray>.

```
#include <valarray>
```

В частности, в заголовочном файле <valarray> определены следующие классы:

```
namespace std {
   template<typename T> class valarray;        // массив значений типа T

   class slice;                                // срез массива значений
   template<typename T> class slice_array;

   class gslice;                               // обобщенный срез
   template<typename T> class gslice_array;

   template<typename T> class mask_array;      // массив значений с маской
   template<typename T> class indirect_array;  // перечисляемый массив значений
}
```

Классы имеют следующий смысл:

- valarray — основной класс, управляющий массивом значений;
- slice и gslice определяют срез как подмножество массива значений по аналогии с библиотекой BLAS;[4]
- slice_array, gslice_array, mask_array и indirect_array — это внутренние вспомогательные классы, используемые для хранения временных значений или данных. Эти классы невозможно использовать для непосредственного программирования интерфейса. Они создаются косвенно определенными операциями над массивами значений.

Все классы параметризованы шаблонным типом элементов. В принципе в качестве шаблонного параметра может использоваться любой тип данных. Однако по природе массивов значений это должен быть числовой тип.

Создание массивов значений

При создании массива значений количество его элементов задается как параметр.

```
std::valarray<int> va1(10);        // массив значений из десяти целых чисел
                                   // с нулевыми значениями
```

[4] Библиотека Basic Linear Algebra Subprograms (BLAS) содержит основные средства для реализации операций линейной алгебры, таких как умножение матриц, решение треугольных систем и простые векторные операции.

```
std::valarray<float> va2(5.7,10);  // массив значений из десяти чисел типа
                                   // float со значением 5.7

                                   // (обратите внимание на порядок аргументов)
```

При передаче одного аргумента он интерпретируется как размер. Элементы инициализируются конструктором типа значений по умолчанию. Элементы массива значений, имеющие фундаментальный тип, инициализируются нулем (см. описание процесса инициализации данных фундаментального типа с помощью конструктора по умолчанию, приведенное в разделе 3.2.1). При передаче второго параметра первый параметр считается начальным значением элемента, а второй — количеством элементов. Отметим, что порядок передачи двух аргументов конструктору отличается от всех других классов в стандартной библиотеке C++. Все контейнерные классы STL используют первый числовой аргумент как количество членов, а второй — как начальное значение.

После принятия стандарта C++11 массив значений можно инициализировать с помощью списка инициализации.

```
std::valarray<int> va3 = { 3, 6, 18, 3, 22 };
```

До принятия стандарта C++11 необходимо было использовать обычный массив из языка C.

```
int array[] = { 3, 6, 18, 3, 22 };
// инициализируем массив значений элементами обычного массива
std::valarray<int> va3(array,sizeof(array)/sizeof(array[0]));

// инициализируем второй элемент из четырех
std::valarray<int> va4(array+1,3);
```

Массив значений копирует передаваемое значение. Таким образом, для инициализации можно передавать временные данные.

Операции над массивами значений

Для доступа к элементу массива значений определен оператор индексации массивов значений. Как обычно, первый индекс равен нулю.

```
va[0] = 3 * va[1] + va[2];
```

Кроме того, для массивов значений определены обычные числовые операции: сложение, вычитание, умножение, деление по модулю, отрицание, побитовые операции, операции сравнения и логические операции, а также все операции присваивания. Эти операции применяются к каждому элементу массива значений. Таким образом, результат операции над массивом значений — это массив значений, размер которого совпадает с размерами операндов, а элементы являются результатами поэлементного вычисления. Например, инструкция

```
va1 = va2 * va3;
```

эквивалентна

```
va1[0] = va2[0] * va3[0];
va1[1] = va2[1] * va3[1];
va1[2] = va2[2] * va3[2];
...
```

Если количество элементов в комбинируемых массивах значений разное, результат становится неопределенным. Разумеется, можно выполнять только те операции, которые предусмотрены для элементов массивов значений. Точный смысл операции зависит от смысла операции над элементами. Таким образом, все эти операции просто делают одно и то же с каждым элементом или парой элементов обрабатываемого массива значений.

В бинарных операциях один из операндов может быть отдельным значением, тип которого совпадает с типом элементов массива. В этом случае отдельное значение комбинируется с каждым элементом массива значений, используемого в качестве второго операнда.

Например, инструкция

```
val = 4 * va2;
```

эквивалентна инструкциям

```
val[0] = 4 * va2[0];
val[1] = 4 * va2[1];
val[2] = 4 * va2[2];
...
```

Тип отдельного значения должен точно совпадать с типом элемента массива значений. Следовательно, предыдущий пример будет работоспособным только в том случае, если элементы имеют тип `int`. Следующая запись оказывается ошибочной:

```
std::valarray<double> va(20);
...
va = 4 * va; // ОШИБКА: несовпадение типов
```

Эта схема бинарных операций применима и к операциям сравнения. Таким образом, операция == *не возвращает* отдельное булево значение, означающее, что массивы значений равны. Вместо этого операция возвращает новый массив значений с тем же количеством элементов типа `bool`, в котором каждый элемент является результатом индивидуального сравнения. Например, в коде

```
std::valarray<double> val(10);
std::valarray<double> va2(10);
std::valarray<bool> vab(10);
...
vab = (val == va2);
```

последняя инструкция эквивалентна записи

```
vab[0] = (val[0] == va2[0]);
vab[1] = (val[1] == va2[1]);
vab[2] = (val[2] == va2[2]);
...
vab[9] = (val[9] == va2[9]);
```

По этой причине невозможно упорядочить массив значений с помощью операции < и нельзя использовать такие массивы, как элементы контейнеров STL, если проверка равенства выполняется с помощью операции == (см. раздел 6.11.1, в котором описаны требования к элементам контейнеров STL).

Следующая программа демонстрирует простое использование массивов значений:

```
// num/valarray1.cpp

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

// выводим массив значений
template <typename T>
void printValarray (const valarray<T>& va)
{
   for (int i=0; i<va.size(); i++) {
        cout << va[i] << ' ';
}
   cout << endl;
}

int main()
{
   // определяем два массива значений с десятью элементами
   valarray<double> val(10), va2(10);

   // присваиваем значения 0.0, 1.1 до 9.9 первому массиву значений
   for (int i=0; i<10; i++) {
        val[i] = i * 1.1;
   }

   // присваиваем -1 всем элементам второго массива значений
   va2 = -1;

   // выводим оба массива значений
   printValarray(val);
   printValarray(va2);

   // выводим минимум, максимум и сумму элементов первого массива значений
   cout << "min(): " << val.min() << endl;
   cout << "max(): " << val.max() << endl;
   cout << "sum(): " << val.sum() << endl;

   // присваиваем значение первого массива значений второму
   va2 = val;

   // удаляем все элементы из первого массива значений
   val.resize(0);

   // снова выводим оба массива значений
   printValarray(val);
   printValarray(va2);
}
```

Программа выводит следующий результат:

```
0 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
min(): 0
```

```
max(): 9.9
sum(): 49.5
0 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9
```

Начиная со стандарта C++11, для обработки элементов можно использовать интервальные циклы `for`, потому что массивы значений имеют глобальные функции `begin()` и `end()` (см. раздел S.2.3).

```
// выводим массив значений (начиная со стандарта C++11):
template <typename T>
void printValarray (const valarray<T>& va)
{
   for (const T& elem: va) {
        cout << elem << ' ';
}
        cout << endl;
}
```

Трансцендентные функции

Трансцендентные функции (тригонометрические и экспоненциальная) определены как эквиваленты числовых операций. Эти операции выполняются над всеми элементами массивов значений. В бинарных операциях один из операндов может быть отдельным значением, а вторым операндом — все значения массива значений.

Все эти операции определены как глобальные функции, а не как функции-члены, чтобы обеспечивать автоматическое преобразование типов для подмножеств массивов значений для обоих операндов (подмножества массивов значений описаны в разделе S.2.2).

Рассмотрим второй пример использования массивов значений. Он демонстрирует использование трансцендентных операций:

```
// num/valarray2.cpp

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

// выводим массив значений
template <typename T>
void printValarray (const valarray<T>& va)
{
   for (int i=0; i<va.size(); i++) {
        cout << va[i] << ' ';
   }
        cout << endl;
}

int main()
{
   // создаем и инициализируем массив значений девятью элементами
   valarray<double> va(9);
   for (int i=0; i<va.size(); i++) {
```

```
        va[i] = i * 1.1;
    }

    // выводим массив значений
    printValarray(va);

    // удваиваем значения в массиве значений
    va *= 2.0;

    // снова выводим массив значений
    printValarray(va);

    // создаем второй массив значений, инициализированный значениями
    // первого массива, увеличенными на 10
    valarray<double> vb(va+10.0);

    // выводим второй массив значений
    printValarray(vb);

    // создаем третий массив значений как результат обработки
    // двух существующих массивов значений
    valarray<double> vc(9);
    vc = sqrt(va) + vb/2.0 - 1.0;

    // выводим третий массив значений
    printValarray(vc);
}
```

Программа выводит следующий результат:

```
0 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8
0 2.2 4.4 6.6 8.8 11 13.2 15.4 17.6
10 12.2 14.4 16.6 18.8 21 23.2 25.4 27.6
4 6.58324 8.29762 9.86905 11.3665 12.8166 14.2332 15.6243 16.9952
```

S.2.2. Подмножества массивов значений

Операция индексации [ ] перегружена для специальных вспомогательных объектов, создаваемых на основе массивов значений. Эти вспомогательные объекты определяют разнообразные подмножества массивов значений, обеспечивая элегантный способ работы с подмножествами массивов значений (с доступом для чтения и записи).

Подмножество массива значений определяется с помощью индекса. Рассмотрим пример:

```
va[std::slice(2,4,3)] // четыре элемента с расстоянием между ними, равным 3,

                      // начиная с индекса 2
va[va>7]              // все элементы со значением, большим 7
```

Если определение подмножества, например `std::slice(2,4,3)` или `va>7`, относится к константному массиву значений, выражение возвращает новый массив значений с соответствующими элементами. Однако, если определение подмножества относится к неконстантному массиву значений, то выражение возвращает временный объект специального

вспомогательного класса массивов значений. Этот временный объект содержит не значения подмножества, а его определение. Таким образом, вычисление значений откладывается до момента, когда потребуется окончательный результат.

Преимущество этого механизма, называемого *отложенным вычислением*, заключается в том, что он не вычисляет временные значения выражений. Это экономит время и память. Кроме того, этот метод поддерживает семантику ссылок. Таким образом, подмножества — это логические множества ссылок на исходные значения. Подмножество можно использовать как левостороннее значение оператора (lvalue). Например, подмножеству массива значений можно присвоить результат умножения двух других подмножеств одного и того же массива значений.

Однако, если элементы подмножества-получателя одновременно используются в подмножестве-источнике, могут возникнуть непредвиденные ситуации. Следовательно, любая операция над массивом значений работает корректно, только если элементы подмножества-получателя и элементы всех подмножеств-источников являются разными.

С помощью изощренных определений подмножеств можно придать массивам значений многомерную семантику. Таким образом, массивы значений можно использовать как многомерные массивы.

Существуют четыре способа определения массивов значений.

1. Срезы.
2. Обобщенные срезы.
3. Подмножества с масками.
4. Перечисляемые подмножества.

Проблемы, связанные с подмножествами массивов значений

Прежде чем перейти к описанию подмножеств, рассмотрим одну общую проблему. Обработка подмножеств массивов значений плохо спроектирована. Создать подмножества легко, но комбинировать их с другими подмножествами трудно. К сожалению, почти всегда требуется явное преобразование типа в класс `valarray`. Причина заключается в том, что стандартная библиотека C++ не гарантирует, что подмножества массивов значений поддерживают те же операции, что и массивы значений.

Например, чтобы умножить два подмножества и присвоить результат третьему подмножеству, нельзя написать код

```
// ОШИБКА: нет преобразования
va[std::slice(0,4,3)] = va[std::slice(1,4,3)] * va[std::slice(2,4,3)];
```

Вместо этого необходимо написать код, использующий приведение

```
va[std::slice(0,4,3)]
   = static_cast<std::valarray<double>>(va[std::slice(1,4,3)]) *
   static_cast<std::valarray<double>>(va[std::slice(2,4,3)]);
```

или приведение в старом стиле:

```
va[std::slice(0,4,3)]
   = std::valarray<double>(va[std::slice(1,4,3)]) *
   std::valarray<double>(va[std::slice(2,4,3)]);
```

Это неудобно и повышает вероятность ошибок. Что еще хуже, без хорошей оптимизации производительность программы может снизиться, потому что каждое приведение создает временный объект, который можно было избежать без приведения.

Для того чтобы обработка была более удобной, можно применить следующую шаблонную функцию:

```
// шаблон для преобразования подмножества массива значений в массив значений
template <typename T>
inline
std::valarray<typename T::value_type> VA (const T& valarray_subset)
{
   return std::valarray<typename T::value_type>(valarray_subset);
}
```

Используя этот шаблон, можно было бы написать следующие инструкции:

```
va[std::slice(0,4,3)] = VA(va[std::slice(1,4,3)]) *
                        VA(va[std::slice(2,4,3)]); // OK
```

Однако проблемы с производительностью все равно остаются.

Если в программе используется определенный тип элементов, можно применить простое определение типа:

```
typedef valarray<double> VAD;
```

С помощью этого определения типа можно также написать операторы

```
va[std::slice(0,4,3)] = VAD(va[std::slice(1,4,3)]) *
                        VAD(va[std::slice(2,4,3)]); // OK
```

при условии, что элементы `va` имеют тип `double`.

Срезы

Срез — это множество индексов, имеющий три следующих свойства.

1. Начальный индекс.
2. Количество элементов (размер).
3. Расстояние между элементами (шаг).

Эти три свойства можно передать точно в таком же порядке, как параметры конструктора класса `slice`. Например, следующее выражение определяет четыре элемента, начиная с индекса `2` и с шагом `3`.

```
std::slice(2,4,3)
```

Иначе говоря, это выражение задает следующий набор индексов:

```
2 5 8 11
```

Шаг может быть отрицательным. Например, выражение

```
std::slice(9,5,-2)
```

задает индексы

```
9 7 5 3 1
```

Для того чтобы определить подмножество массива значений, можно использовать срез как аргумент операции индексации. Например, следующее выражение определяет подмножество массива значений va, содержащее элементы с индексами 2, 5, 8 и 11.

```
va[std::slice(2,4,3)]
```

Вызывающая сторона должна гарантировать, что все эти индексы являются корректными.

Если подмножество квалифицировано как срез константного массива значений, то оно считается новым массивом значений. Если массив значений не является константным, то подмножество содержит ссылки на исходный массив значений. Вспомогательный класс slice_array используется следующим образом:

```
namespace std {
   class slice;

   template <typename T>
   class slice_array;

   template <typename T>
   class valarray {
      public:
      // срез константного массива значений возвращает новый массив значений
      valarray<T> operator[] (slice) const;

      // срез переменного числового массива возвращает объект класса slice_array
      slice_array<T> operator[] (slice);
      ...
   };
}
```

Для объектов класса slice_array определены следующие операции.

- Присваивание отдельного значения всем элементам.
- Присваивание другого массива значений (или подмножества массива значений).
- Вызов любой вычисляемой операции присваивания, например += или *=.

Для любой другой операции требуется преобразование подмножества в массив значений (см. раздел S.2.2). Класс slice_array<> является внутренним вспомогательным классом для срезов и должен быть доступным для пользователя. Таким образом, все конструкторы и операции присваивания в классе slice_array<> являются закрытыми.

Например, оператор

```
va[std::slice(2,4,3)] = 2;
```

присваивает 2 третьему, шестому, девятому и двенадцатому элементам массива значений va. Это эквивалентно следующим операторам:

```
va[2] = 2;
va[5] = 2;
va[8] = 2;
va[11] = 2;
```

В качестве другого примера рассмотрим следующий оператор, который возводит в квадрат значения элементов с индексами 2, 5, 8 и 11.

```
va[std::slice(2,4,3)]


   *= std::valarray<double>(va[std::slice(2,4,3)]);
```

Как указано в разделе S.2.2, нельзя просто написать

```
va[std::slice(2,4,3)] *= va[std::slice(2,4,3)]; // ОШИБКА
```

Однако, используя шаблонную функцию `VA()`, упомянутую в разделе S.2.2, можно написать

```
va[std::slice(2,4,3)] *= VA(va[std::slice(2,4,3)]); // OK
```

Передавая разные срезы одного и того же массива значений, можно комбинировать разные подмножества и сохранять результаты в других подмножествах массива значений. Например, выражение

```
va[std::slice(0,4,3)] = VA(va[std::slice(1,4,3)]) *
                        VA(va[std::slice(2,4,3)]);
```

эквивалентно следующим инструкциям:

```
va[0] = va[1]  * va[2];
va[3] = va[4]  * va[5];
va[6] = va[7]  * va[8];
va[9] = va[10] * va[11];
```

Если рассматривать массив значений как двумерную матрицу, то этот пример представляет собой умножение векторов (рис. S.1). Однако порядок отдельных операций присваивания не определен. Следовательно, поведение становится неопределенным, если подмножество-получатель содержит элементы, используемые в подмножествах-источниках.

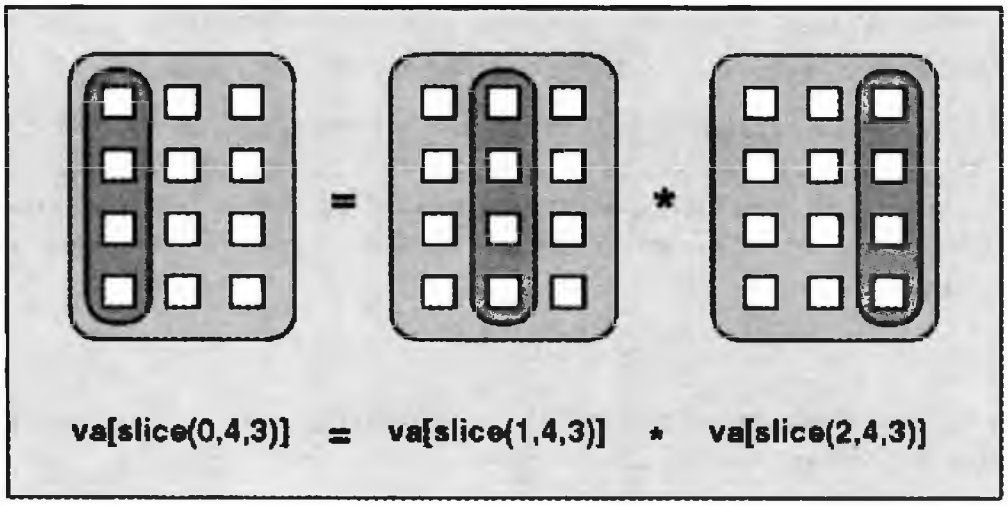


Рис. S.1. Умножение срезов массива значений

Таким способом можно создавать еще более сложные выражения:

```
va[std::slice(0,100,3)]
   = std::pow(VA(va[std::slice(1,100,3)]) * 5.0,
              VA(va[std::slice(2,100,3)]));
```

Отметим еще раз, что отдельное значение, в данном случае `5.0`, должно точно соответствовать типу элементов массива значений

Следующая программа демонстрирует законченный пример использования срезов массивов значений:

```
// num/slice1.cpp

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

// построчный вывод массива значений
template <typename T>
void printValarray (const valarray<T>& va, int num)
{
   for (int i=0; i<va.size()/num; ++i) {
      for (int j=0; j<num; ++j) {
         cout << va[i*num+j] << ' ';
      }
      cout << endl;
   }
   cout << endl;
}
int main()
{
   // массив значений из 12 элементов
   // - четыре строки
   // - три столбца
   valarray<double> va(12);

   // заполняем массив значений значениями
   for (int i=0; i<12; i++) {
        va[i] = i;
   }

   printValarray (va, 3);

   // первая строка = второй столбец, возведенный в степень,
   // стоящую в третьем столбце
   va[slice(0,4,3)] = pow (valarray<double>(va[slice(1,4,3)]),
                           valarray<double>(va[slice(2,4,3)]));
                           printValarray (va, 3);

   // создаем числовой массив, в котором трижды повторяется третий элемент
   // массива значений  va
   valarray<double> vb(va[slice(2,4,0)]);
```

```
    // умножаем третий столбец на элементы массива значений vb
    va[slice(2,4,3)] *= vb;

    printValarray (va, 3);

    // выводим квадратный корень из элементов второй строки
    printValarray (sqrt(valarray<double>(va[slice(3,3,1)])));

    // удваиваем элементы третьей строки
    va[slice(2,4,3)] = valarray<double>(va[slice(2,4,3)]) * 2.0;

    printValarray (va, 3);
}
```

Программа выводит следующие результаты:

```
0 1 2
3 4 5
6 7 8
9 10 11

1 1 2
1024 4 5
5.7648e+006 7 8
1e+011 10 11

1 1 4
1024 4 10
5.7648e+006 7 16
1e+011 10 22
32 2 3.16228

1 1 8
1024 4 20
5.7648e+006 7 32
1e+011 10 44
```

Обобщенные срезы

Обобщенные срезы (general slices, или *gslices*) — это универсальная форма срезов. Подобно обычным срезам, позволяющим обрабатывать одномерное подмножество, обобщенные срезы позволяют работать с подмножествами многомерных массивов. В принципе обобщенные срезы имеют те же свойства, что и обычные срезы.

- Начальный индекс.
- Количество элементов (размер).
- Расстояние между элементами (шаг).

Тем не менее, в отличие от обычных срезов, в обобщенных срезах количество элементов и шаг являются массивами значений. Количество элементов в таком массиве равно количеству используемых размерностей. Например, обобщенный срез, имеющий состояние

```
start: 2
size: [ 4 ]
stride: [ 3 ]
```[5]

эквивалентен обычному срезу, потому что этот массив имеет одну размерность. Таким образом, обобщенные срезы определяют четыре элемента с шагом 3, начиная с индекса 2:

```
2 5 8 11
```

Однако обобщенный срез, имеющий состояние

```
start: 2
size: [ 2 4 ]
stride: [ 10 3 ]
```

имеет две размерности. Наименьший индекс имеет высшую размерность. Таким образом, этот обобщенный срез задает начало с индексом 2, два элемента с шагом 10 и четыре элемента с шагом 3:

```
2 5 8 11
12 15 18 21
```

Рассмотрим пример среза с тремя размерностями:

```
start: 2
size: [ 3 2 4 ]
stride: [ 30 10 3 ]
```

Он задает начало с индексом 2, три элемента с шагом 30, два элемента с шагом 10 и четыре элемента с шагом 3:

```
 2  5  8 11
12 15 18 21

32 35 38 41
42 45 48 51

62 65 68 71
72 75 78 81
```

Единственное отличие обобщенных срезов от обычных — это возможность использовать массивы для определения размеров и шагов. Помимо этого, обобщенные срезы ничем не отличаются от обычных срезов.

1. Для определения конкретного подмножества массива значений достаточно передать обобщенный срез операции индексации массива значений.
2. Если массив значений является константным, то результатом выражения является новый массив значений.
3. Если массив значений является неконстантным, то результирующим выражением является объект класса `gslice_array`, представляющий элементы массива значений с семантикой ссылок:

[5] Stride — расстояние между элементами. — *Примеч. ред.*

```
namespace std {
   class gslice;

   template <typename T>
   class gslice_array;

   template <typename T>
   class valarray {
      public:
         // обобщенный срез константного массива значений
         // возвращает новый массив значений
         valarray<T> operator[] (const gslice&) const;

         // обобщенный срез переменного массива значений
         // возвращает объект класса gslice_array
         gslice_array<T> operator[] (const gslice&);
         ...
   };
}
```

4. Для объектов класса `gslice_array` предусмотрены присваивание и вычисляемые операции присваивания, модифицирующие элементы подмножества.
5. С помощью преобразования типов можно комбинировать обобщенный срез с другими массивами значений и подмножествами массивов значений (см. раздел S.2.2).

Использование обобщенных срезов массивов значений демонстрирует следующая программа:

```
// num/gslice1.cpp

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

// построчный вывод трехмерного массива значений
template <typename T>
void printValarray3D (const valarray<T>& va, int dim1, int dim2)
{
   for (int i=0; i<va.size()/(dim1*dim2); ++i) {
      for (int j=0; j<dim2; ++j) {
         for (int k=0; k<dim1; ++k) {
            cout << va[i*dim1*dim2+j*dim1+k] << ' ';
         }
         cout << '\n';
      }
      cout << '\n';
   }
   cout << endl;
}

int main()
{
   // массив значений из 24 элементов
   // - две группы
```

```
    // - четыре строки
    // - три столбца
    valarray<double> va(24);

    // заполняем массив значениями
    for (int i=0; i<24; i++) {
       va[i] = i;
    }

    // выводим массив значений
    printValarray3D (va, 3, 4);

    // нам необходимы два двумерных подмножества, в которых
    // записаны три тройки в двух 12-элементых массивах
    size_t lengthvalues[] = { 2, 3 };
    size_t stridevalues[] = { 12, 3 };
    valarray<size_t> length(lengthvalues,2);
    valarray<size_t> stride(stridevalues,2);

    // присваиваем второй столбец первых трех строк
    // первому столбцу первых трех строк
    va[gslice(0,length,stride)]
       = valarray<double>(va[gslice(1,length,stride)]);

    // добавляем и присваиваем третью из первых трех строк
    // первой из первых трех строк
    va[gslice(0,length,stride)]
       += valarray<double>(va[gslice(2,length,stride)]);

    // выводим массив значений
    printValarray3D (va, 3, 4);
}
```

Программа выводит следующие результаты:

```
0 1 2
3 4 5
6 7 8
9 10 11

12 13 14
15 16 17
18 19 20
21 22 23

3 1 2
9 4 5
15 7 8
9 10 11

27 13 14
33 16 17
39 19 20
21 22 23
```

Маскированные подмножества

Еще один способ определить подмножество массива значений обеспечивают маскированные массивы. Накладывать маску на элементы можно с помощью булевых выражений. Например, в выражении

```
va[va > 7]
```

подвыражение

```
va > 7
```

определяет массив значений размера va, в котором булево значение указывает, имеет ли данный элемент значение, большее 7. Операция индексации использует булев массив значений для определения всех элементов, для который булево выражение возвращает значение true. Таким образом, в массиве значений va выражение

```
va[va > 7]
```

задает подмножество элементов, которые больше 7.

Помимо этого, маскированные массивы ведут себя точно так же, как и все остальные подмножества массива значений.

1. Для определения конкретного подмножества массива значений достаточно передать массив значений булевых значений операции индексации массива значений.
2. Если массив значений является константным, то результатом выражения становится новый массив значений.
3. Если массив значений является неконстантным, то результатом выражения является объект класса mask_array, представляющий элементы массива значений с семантикой ссылок:

```
namespace std {
   template <typename T>
   class mask_array;

   template <typename T>
   class valarray {
      public:
        // наложение маски на константный массив значений
        valarray<T> operator[] (const valarray<bool>&) const;

        // masking a variable valarray returns a mask_array
        mask_array<T> operator[] (const valarray<bool>&);
        ...
   };
}
```

4. Для объектов класса mask_array предусмотрены присваивание и вычисляемые операции присваивания, модифицирующие элементы подмножества.
5. С помощью преобразования типов можно комбинировать маскированные массивы и другие массивы значений, а также подмножества массивов значений (см. раздел S.2.2).

Использование маскированных массивов демонстрирует следующая программа:

```
// num/maskarray1.cpp

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

// построчный вывод массива значений
template <typename T>
void printValarray (const valarray<T>& va, int num)
{
   for (int i=0; i<va.size()/num; ++i) {
      for (int j=0; j<num; ++j) {
         cout << va[i*num+j] << ' ';
      }
      cout << endl;
   }
   cout << endl;
}

int main()
{
   // массив значений из 12 элементов
   // - четыре строки
   // - три столбца
   valarray<double> va(12);

   // заполняем массив значений
   for (int i=0; i<12; i++) {
      va[i] = i;
   }

   printValarray (va, 3);

   // присваиваем 77 всем значениям, которые меньше 5
   va[va<5.0] = 77.0;

   // добавляем 100 ко всем значениям, которые больше 5 и меньше 9
   va[va>5.0 && va<9.0]
      = valarray<double>(va[va>5.0 && va<9.0]) + 100.0;

   printValarray (va, 3);
}
```

Программа выводит следующий результат:

```
0 1 2
3 4 5
6 7 8
9 10 11

77 77 77
```

```
77 77 5
106 107 108
9 10 11
```

Типы сравниваемых числовых значений из маскированных массивов должны точно совпадать. Например, значение `int`, которое используется для сравнения со значениями типа `double` из массива значений, является неприемлемым:

```
valarray<double> va(12);
...
va[va<5] = 77; // ОШИБКА
```

Перечисляемые подмножества

Перечисляемые массивы обеспечивают четвертый, и последний, способ определения подмножества массива значений. В данном случае можно просто определить подмножество массива значений, передав массив индексов. Индексы, задающие подмножество, могут быть не упорядоченными и указываться дважды.

Во всем остальном перечисляемые массивы ничем не отличаются от обычных подмножеств.

1. Для определения конкретного подмножества массива значений достаточно передать массив элементов типа `size_t` операции индексации массива значений.
2. Если массив значений является константным, то результатом выражения является новый массив значений.
3. Если массив значений является неконстантным, то результатом выражения является объект класса `indirect_array`, представляющий элементы массива значений с семантикой ссылок.

```
namespace std {
   template <typename T>
   class indirect_array;

   template <typename T>
   class valarray {
      public:
         // операция индексации константного массива значений возвращает
         // новый массив значений
         valarray<T> operator[] (const valarray<size_t>&) const;

         // операция индексации переменного массива значений возвращает
         // объект класса indirect_array
         indirect_array<T> operator[] (const valarray<size_t>&);
         ...
   };
}
```

4. Для объектов класса `indirect_array` предусмотрены присваивание и вычисляемые операции присваивания, модифицирующие элементы подмножества.
5. С помощью преобразования типов можно комбинировать перечислимые массивы и другие массивы значений, а также подмножества массивов значений (см. раздел S.2.2).

Использование перечислимых массивов демонстрирует следующая программа:

```
// num/indirectarray1.cpp

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

// выводим массив значений как двумерный массив
template <typename T>
void printValarray (const valarray<T>& va, int num)
{
   for (int i=0; i<va.size()/num; i++) {
      for (int j=0; j<num; j++) {
         cout << va[i*num+j] << ' ';
      }
      cout << endl;
   }
   cout << endl;
}

int main()
{
   // создаем массив значений из 12 элементов
   valarray<double> va(12);

   // инициализируем массив значений числами 1.01, 2.02, ... 12.12
   for (int i=0; i<12; i++) {
      va[i] = (i+1) * 1.01;
   }

   printValarray(va,4);

   // создаем массив индексов
   // - примечание: элементы должны иметь тип size_t
   valarray<size_t> idx(4);
   idx[0] = 8;
   idx[1] = 0;
   idx[2] = 3;
   idx[3] = 7;

   // используем массив значений для вывода девятого, первого,
   // четвертого и восьмого элементов
   printValarray(valarray<double>(va[idx]), 4);

   // изменяем первый и четвертый элементы и выводим их косвенным образом
   va[0] = 11.11;
   va[3] = 44.44;
   printValarray(valarray<double>(va[idx]), 4);

   // теперь выбираем второй, третий, шестой и девятый элементы
   // и присваиваем им 99
   idx[0] = 1;
```

```
    idx[1] = 2;
    idx[2] = 5;
    idx[3] = 8;
    va[idx] = 99;

    // снова выводим весь массив значений
    printValarray (va, 4);
}
```

Массив значений `idx` используется для определения подмножества элементов в массиве значений `va`. Программа выводит следующие результаты:

```
1.01 2.02 3.03 4.04
5.05 6.06 7.07 8.08
9.09 10.1 11.11 12.12

9.09 1.01 4.04 8.08

9.09 11.11 44.44 8.08

11.11 99 99 44.44
5.05 99 7.07 8.08
99 10.1 11.11 12.12
```

S.2.3. Подробное описание класса `valarray`

В основе массивов значений лежит шаблонный класс `valarray<>`. Он параметризован типом элементов:

```
namespace std {
    template <typename T>
    class valarray;
}
```

Размер не является частью типа. Следовательно, в принципе можно обрабатывать массивы значений разных размеров, а также изменять эти размеры. Однако изменение размера массива значений предназначено только для реализации двухэтапной инициализации (создание и установка размера), необходимой для управления массивами массивов значений. Результат комбинации массивов значений с разными размерами не определен.

Создание, копирование и уничтожение

***valarray*::valarray ()**

- Конструктор по умолчанию.
- Создает пустой массив значений.
- Этот конструктор предназначен только для создания массивов значений. На следующем этапе их правильный размер задается функцией-членом `resize()`.

`explicit` **valarray::valarray** (`size_t` *num*)

- Создает массив значений, содержащий *num* элементов.

- Элементы инициализируются их конструкторами по умолчанию. Элементы фундаментальных типов инициализируются нулями 0.

***valarray*::valarray** (*initializer-list*)

- Создает массив значений, содержащий значения из списка *инициализации*.
- Начиная со стандарта C++11

***valarray*::valarray** (`const T&` *value*, `size_t` *num*)

- Создает массив значений, содержащий *num* элементов.
- Элемент инициализируется значением *value*.
- Порядок параметров необычен. Все другие классы в стандартной библиотеке C++ имеют интерфейс, в котором значение *num* является первым параметром, а значение *value* — вторым.

***valarray*::valarray** (`const T*` *array*, `size_t` *num*)

- Создает массив значений, содержащий *num* элементов.
- Элементы инициализируются значениями элементом массива *array*.
- Вызывающая сторона должна гарантировать, что массив *array* содержит *num* элементов; в противном случае поведение не определено.

***valarray*::valarray** (`const` *valarray*`&` *va*)
***valarray*::valarray** (*valarray*`&&` *va*)

- Копирующие и перемещающий конструкторы.
- Создает массив значений как копию массива *va* или как массив, содержащий перемещенные элементы массива *va*.
- Перемещающий конструктор предусмотрен стандартом C++11.

***valarray*::~valarray** ()

- Деструктор.
- Уничтожает все элементы и освобождает память.

Можно также создать массивы значений, инициализированные объектами внутренних вспомогательных классов `slice_array`, `gslice_array`, `mask_array` и `indirect_array`.

Операции присваивания

valarray`&` ***valarray*::operator =** (`const` *valarray*`&` *va*)
valarray`&` ***valarray*::operator =** (*valarray*`&&` *va*)

- Операция копирующего или перемещающего присваивания элементов массива значений *va*.
- Если массив *va* имеет другой размер, поведение не определено.
- Значение элемента в левой части операции присваивания массивов значений не должно зависеть от значения другого элемента в левой части. Иначе говоря, если присваивание перезаписывает значения, используемые в правой части присваивания,

то результат будет не определен. Это значит, что не следует использовать элемент из левой части в выражении правой части. Это объясняется тем, что порядок вычисления операторов над массивами значений не определен (см. раздел S.2.2).

- Перемещающая операция присваивания предусмотрена стандартом C++11.

`void` ***valarray*`::`swap** (*valarray& va2*)
`void` **swap** (*valarray& va1, valarray& va2*)

- Обе версии обменивают значения двух массивов значений.
  - Функция-член обменивает содержимое массивов `*this` и *va2*.
  - Глобальная функция обменивает содержимое массивов *va1* и *va2*.
- По возможности следует выбирать эти функции, а не операции присваивания, потому что они выполняются быстрее. Фактически они имеют константную сложность.
- Предусмотрены стандартом C++11.

valarray& ***valarray*`::`operator** = (*initializer-list*)

- Присваивает значения из *списка инициализации*.
- Возвращает `*this`.
- Начиная со стандарта C++11.

valarray& **valarray`::`operator** = (`const T&` *value*)

- Присваивает значение *value* каждому элементу массива значений.
- Размер массива значений не изменяется. Указатели и ссылки на элементы остаются корректными.

Кроме того, можно присваивать объекты внутренних вспомогательных классов `slice_array`, `gslice_array`, `mask_array` и `indirect_array`.

Функции-члены

В класс `valarray` входят следующие функции-члены:

`size_t` ***valarray*`::`size** () `const`

- Возвращает текущее количество элементов.

`void` ***valarray*`::`resize** (`size_t` *num*)
`void` ***valarray*`::`resize** (`size_t` *num*, `T` *value*)

- Обе версии изменяют размер массива значений *num*.
- Если размер увеличивается, то новые элементы инициализируются их конструкторами по умолчанию или значением *value* соответственно.
- Обе версии делают некорректными все указатели и ссылки на элементы массива значений.
- Эти функции предназначены только для создания массивов значений. После их создания с помощью конструктора по умолчанию необходимо установить их правильные размеры, вызвав указанные функции.

T ***valarray***::**min** () const
T ***valarray***::**max** () const

- Первая версия возвращает минимальное значение всех элементов.
- Вторая версия возвращает максимальное значение всех элементов.
- Элементы сравниваются с помощью операции < или > соответственно. Таким образом, эти операции должны быть предусмотрены типом элементов.
- Если массив значений не содержит элементов, то возвращаемое значение не определено.

T ***valarray***::**sum** () const

- Возвращает сумму всех элементов.
- Элементы обрабатываются с помощью операции +=. Таким образом, эта операция должна быть предусмотрена типом элементов.
- Если массив значений не содержит элементов, то возвращаемое значение не определено.

valarray ***valarray***::**shift** (int *num*) const

- Возвращает новый массив значений, в котором все элементы сдвинуты на *num* позиций.
- Возвращаемый массив значений имеет такое же количество элементов.
- Элементы на сдвинутых позициях инициализируются их конструкторами по умолчанию.
- Направление сдвига зависит от знака параметра *num*.
  - Если *num* — положительное число, то сдвиг выполняется влево, к началу. Таким образом, элементы получают меньшие индексы.
  - Если *num* — отрицательное число, то сдвиг выполняется вправо, к концу. Таким образом, элементы получают бо́льшие индексы.

valarray ***valarray***::**cshift** (int *num*) const

- Возвращает новый массив значений, в котором все элементы циклически сдвинуты на *num* позиций.
- Возвращаемый массив значений имеет такое же количество элементов.
- Направление сдвига зависит от знака параметра *num*.
  - Если *num* — положительное число, то сдвиг выполняется влево, к началу. Таким образом, элементы получают меньшие индексы или вставляются в конец массива.
  - Если *num* — отрицательное число, то сдвиг выполняется вправо, к концу. Таким образом, элементы получают бо́льшие индексы или вставляются в начало.

valarray ***valarray***::**apply** (T *op*(T)) const
valarray ***valarray***::**apply** (T *op*(const T&)) const

- Обе версии возвращают новый массив значений, в котором ко всем элементам применена операция *op*().

- Возвращаемый массив значений имеет такое же количество элементов.
- Для каждого элемента массива `*this` вызывается операция *op*(*elem*), инициализирующая соответствующий элемент в новом возвращаемом массиве значений ее результатом.

Доступ к элементам

T`&` ***valarray*`::operator[ ] (size_t` *idx*`)`
`const` *T*`&` ***valarray*`::operator[ ] (size_t` *idx*`) const`

- Обе версии возвращают элемент массива значений с индексом *idx* (первый элемент имеет индекс `0`).
- Неконстантная версия возвращает ссылку. Таким образом, можно модифицировать элемент, заданный и возвращенный этой операцией. Гарантируется, что ссылка является корректной, пока массив значений существует и не вызываются функции, изменяющие его размер.
- До принятия стандарта C++11 значение, возвращаемое второй версией, имело тип *T*.

iterator **begin** `(`*valarray*`& va)`
const_iterator **begin** `(const` *valarray*`& va)`

- Обе версии возвращают итератор произвольного доступа, установленный на начало массива значений (на позицию первого элемента).
- Имя типа итератора не определено.
- Если контейнер пустой, то вызовы этих функций эквивалентны вызову *valarray*`::end()`.
- Начиная со стандарта C++11

iterator **end** `(`*valarray*`& va)`
const_iterator **end** `(const` *valarray*`& va)`

- Обе версии возвращают итератор произвольного доступа, установленный на конец контейнера (на позицию последнего элемента).
- Имя типа итератора не определено.
- Если контейнер пустой, то вызовы этих функций эквивалентны вызову *valarray*`::begin()`.
- Начиная со стандарта C++11.

Операции над массивами значений

Унарные операции над массивами значений имеют следующий формат:

valarray ***valarray*`::` **unary-op** `() const`

- Унарная операция возвращает новый массив значений, содержащий все значения массива `*this`, модифицированные операцией **unary-op**.
- Операцией **unary-op** может быть одна из следующих операций:

```
operator +
operator -
operator ~
operator !
```

- Значение, возвращаемое операцией `!`, имеет тип `valarray<bool>`.

Бинарные операции над массивами значений (за исключением операций сравнения и присваивания) имеют следующий формат:

valarray ***binary-op*** `(const` *valarray*`&` *va1*`, const` *valarray*`&` *va2*`)`
valarray ***binary-op*** `(const` *valarray*`&` *va*`, const T&` *value*`)`
valarray ***binary-op*** `(const T&` *value*`, const` *valarray*`&` *va*`)`

- Эти операции возвращают новый массив значений с тем же количеством элементов, что и `va`, `va1` или `va2`. Новый массив значений содержит результат применения операции ***binary-op*** к каждой паре значений.
- Если передается только один операнд в виде единственного значения *value*, то этот операнд комбинируется с каждым элементом массива *va*.
- Операцией ***binary-op*** может быть одна из следующих операций:

```
operator +
operator -
operator *
operator /
operator %
operator ^
operator &
operator |
operator <<
operator >>
```

- Если массивы *va1* и *va2* имеют разное количество элементов, то результат не определен.

Логические операции и операции сравнения имеют аналогичную структуру. Однако они возвращают значения булева типа.

`valarray<bool>` ***logical-op*** `(const` *valarray*`&` *va1*`, const` *valarray*`&` *va2*`)`
`valarray<bool>` ***logical-op*** `(const` *valarray*`&` *va*`, const T&` *value*`)`
`valarray<bool>` ***logical-op*** `(const T&` *value*`, const` *valarray*`&` *va*`)`

- Эти операции возвращают новый массив значений с тем же количеством элементов, что и `va`, `va1` или `va2`. Новый массив значений содержит результат применения операции **logical-op** к каждой паре значений.
- Если передается только один операнд в виде единственного значения *value*, то он комбинируется с каждым элементом массива *va*.
- Операцией **logical-op** может быть одна из следующих операций:

```
operator ==
operator !=
operator <
operator <=
operator >
```

```
operator >=
operator &&
operator ||
```

- Если массивы *va1* и *va2* имеют разное количество элементов, то результат не определен.

Для массивов значений предусмотрены вычисляемые операции присваивания.

valarray& ***valarray*::*assign-op*** (const *valarray*& *va*)
valarray& ***valarray*::*assign-op*** (const T& *value*)

- Обе версии применяют операцию ***assign-op*** к каждому элементу массива `*this` и соответствующему элементу массива *va* или *value* соответственно, переданному как второй операнд.
- Функции возвращают ссылку на модифицированный массив значений.
- Операцией ***assign-op*** может быть одна из следующих операций:

```
operator +=
operator -=
operator *=
operator /=
operator %=
operator &=
operator |=
operator ^=
operator <<=
operator >>=
```

- Если массивы `*this` и *va2* содержат разное количество элементов, то результат не определен.
- Ссылки и указатели на модифицированные элементы остаются корректными, пока массив значений существует и не вызываются функции, изменяющие его размер.

Трансцендентные функции

valarray **abs** (const *valarray*& *va*)
valarray **pow** (const *valarray*& *va1*, const *valarray*& *va2*)
valarray **pow** (const *valarray*& *va*, const T& *value*)
valarray **pow** (const T& *value*, const *valarray*& *va*)
valarray **exp** (const *valarray*& *va*)
valarray **sqrt** (const *valarray*& *va*)
valarray **log** (const *valarray*& *va*)
valarray **log10** (const *valarray*& *va*)
valarray **sin** (const *valarray*& *va*)
valarray **cos** (const *valarray*& *va*)
valarray **tan** (const *valarray*& *va*)
valarray **sinh** (const *valarray*& *va*)
valarray **cosh** (const *valarray*& *va*)
valarray **tanh** (const *valarray*& *va*)
valarray **asin** (const *valarray*& *va*)

valarray **acos** (const *valarray*& *va*)
valarray **atan** (const *valarray*& *va*)
valarray **atan2** (const *valarray*& *val*, const *valarray*& *va2*)
valarray **atan2** (const *valarray*& *va*, const T& *value*)
valarray **atan2** (const T& *value*, const *valarray*& *va*)

- Все эти функции возвращают новый массив значений с тем же количеством элементов, что и va, val и va2. Новый массив значений содержит результат соответствующей операции, примененной к каждому элементу или паре элементов.
- Если массивы *val* и *va2* имеют разное количество элементов, то результат не определен.

S.2.4. Подробное описание классов подмножеств массивов значений

Этот подраздел посвящен подробному описанию классов подмножеств массивов значений. Эти классы очень просты и не содержат много операций. По этой причине в подразделе приводятся только объявления и несколько замечаний.

Классы slice и slice_array

Объекты класса slice_array создаются с помощью класса slice, используемого как индекс неконстантного массива значений.

```
namespace std {
   template <typename T>
   class valarray {
      public:
         ...
         slice_array<T> operator[](slice);
         ...
   };
}
```

Точное определение открытого интерфейса класса slice имеет следующий вид:

```
namespace std {
   class slice {
      public:
         slice (); // пустое подмножество
         slice (size_t start, size_t size, size_t stride);
         size_t start() const;
         size_t size() const;
         size_t stride() const;
   };
}
```

Конструктор по умолчанию создает пустое подмножество. С помощью функций-членов start(), size() и stride() можно определить свойства среза.

Класс slice_array имеет следующие операции:

```
namespace std {
   template <typename T>
   class slice_array {
      public:
         typedef T value_type;

         void operator= (const T&);
         void operator= (const valarray<T>&) const;
         void operator*= (const valarray<T>&) const;
         void operator/= (const valarray<T>&) const;
         void operator%= (const valarray<T>&) const;
         void operator+= (const valarray<T>&) const;
         void operator-= (const valarray<T>&) const;
         void operator^= (const valarray<T>&) const;
         void operator&= (const valarray<T>&) const;
         void operator|= (const valarray<T>&) const;
         void operator<<=(const valarray<T>&) const;
         void operator>>=(const valarray<T>&) const;
         ~slice_array();
      private:
         slice_array();

         slice_array(const slice_array&);
         slice_array& operator=(const slice_array&);
         ...
   };
}
```

Класс `slice_array<>` предназначен исключительно для внутренних вспомогательных целей и должен быть доступен пользователю. Таким образом, все конструкторы и операция присваивания класса `slice_array<>` являются закрытыми.

Классы gslice и gslice_array

Объекты класса `gslice_array` создаются с помощью класса `gslice`, используемого в качестве индекса неконстантного массива значений.

```
namespace std {
   template <typename T>
   class valarray {
      public:
      ...
      gslice_array<T> operator[](const gslice&);
      ...
   };
}
```

Точное определение открытого интерфейса класса `gslice` имеет следующий вид:

```
namespace std {
   class gslice {
      public:
         gslice ();                    // пустое подмножество
```

```
        gslice (size_t start,
                const valarray<size_t>& size,
                const valarray<size_t>& stride);
        size_t start() const;
        valarray<size_t> size() const;
        valarray<size_t> stride() const;
    };
}
```

Конструктор по умолчанию создает пустое подмножество. С помощью функций-членов `start()`, `size()` и `stride()` можно определить свойства обобщенного среза.

Класс `gslice_array` содержит следующие операции:

```
namespace std {
    template <typename T>
    class gslice_array {
        public:
            typedef T value_type;

            void operator= (const T&);
            void operator= (const valarray<T>&) const;
            void operator*= (const valarray<T>&) const;
            void operator/= (const valarray<T>&) const;
            void operator%= (const valarray<T>&) const;
            void operator+= (const valarray<T>&) const;
            void operator-= (const valarray<T>&) const;
            void operator^= (const valarray<T>&) const;
            void operator&= (const valarray<T>&) const;
            void operator|= (const valarray<T>&) const;
            void operator<<=(const valarray<T>&) const;
            void operator>>=(const valarray<T>&) const;
            ~gslice_array();
        private:
            gslice_array();
            gslice_array(const gslice_array<T>&);
            gslice_array& operator=(const gslice_array<T>&);
            ...
    };
}
```

Как и класс `slice_array<>`, класс `gslice_array<>` предназначен исключительно для вспомогательных внутренних целей и должен быть доступен пользователю. Таким образом, все конструкторы и операция присваивания в классе `gslice_array<>` являются закрытыми.

Класс `mask_array`

Объекты класса `mask_array` создаются с помощью класса `valarray<bool>`, используемого в качестве индекса неконстантного массива значений.

```
namespace std {
    template <typename T>
    class valarray {
        public:
```

```
        ...
        mask_array<T> operator[](const valarray<bool>&);
        ...
    };
}
```

Класс `mask_array` имеет следующие операции:

```
namespace std {
   template <typename T>
   class mask_array {
      public:
         typedef T value_type;

         void operator= (const T&);
         void operator= (const valarray<T>&) const;
         void operator*= (const valarray<T>&) const;
         void operator/= (const valarray<T>&) const;
         void operator%= (const valarray<T>&) const;
         void operator+= (const valarray<T>&) const;
         void operator-= (const valarray<T>&) const;
         void operator^= (const valarray<T>&) const;
         void operator&= (const valarray<T>&) const;
         void operator|= (const valarray<T>&) const;
         void operator<<=(const valarray<T>&) const;
         void operator>>=(const valarray<T>&) const;
         ~mask_array();
      private:
         mask_array();
         mask_array(const mask_array<T>&);
         mask_array& operator=(const mask_array<T>&);
         ...
   };
}
```

Как и предыдущие классы, класс `gslice_array<>` предназначен исключительно для вспомогательных внутренних целей и должен быть доступен пользователю. Таким образом, все конструкторы и операция присваивания в классе `mask_array<>` являются закрытыми.

Класс `indirect_array`

Объекты класса `indirect_array` создаются с помощью класса `valarray<size_t>`, используемого в качестве индекса неконстантного массива значений.

```
namespace std {
   template <typename T>
   class valarray {
      public:
         ...
         indirect_array<T> operator[](const valarray<size_t>&);
         ...
      };
}
```

Класс `indirect_array` имеет следующие операции:

```
namespace std {
   template <typename T>
   class indirect_array {
      public:
         typedef T value_type;

         void operator= (const T&);
         void operator= (const valarray<T>&) const;
         void operator*= (const valarray<T>&) const;
         void operator/= (const valarray<T>&) const;
         void operator%= (const valarray<T>&) const;
         void operator+= (const valarray<T>&) const;
         void operator-= (const valarray<T>&) const;
         void operator^= (const valarray<T>&) const;
         void operator&= (const valarray<T>&) const;
         void operator|= (const valarray<T>&) const;
         void operator<<=(const valarray<T>&) const;
         void operator>>=(const valarray<T>&) const;
         ~indirect_array();
      private:
         indirect_array();
         indirect_array(const indirect_array<T>&);
         indirect_array& operator=(const indirect_array<T>&);
         ...
   };
}
```

Как и предыдущие классы, класс `indirect_array<>` предназначен исключительно для вспомогательных внутренних целей и должен быть доступен пользователю. Таким образом, все конструкторы и операция присваивания в классе `indirect_array<>` являются закрытыми.

S.3. Подробное описание распределителей памяти и функций для работы с памятью

Как указано в разделе 4.6 и главе 19, распределители памяти представляют собой специальную модель памяти и являются абстракцией, используемой для перевода *потребности* в памяти в непосредственный *запрос* на выделение памяти. В разделе описываются детали использования распределителей памяти и управления памятью и, в частности, рассматриваются следующие темы.

- Распределители памяти с ограниченной областью видимости.
- Интерфейс распределителей памяти и распределитель памяти по умолчанию.
- Создание распределителей памяти до принятия стандарта C++11.
- Утилиты для неинициализированной памяти.

S.3.1. Распределители памяти с ограниченной областью видимости

В стандарте C++98 концепция распределителей памяти сопровождалась рядом ограничений.

- Контейнерные объекты не обязаны были содержать свои распределители памяти в качестве членов. Распределители памяти были всего лишь частью типа. По этой причине имели место следующие ограничения.
  - Распределители памяти, имеющие одинаковый тип, предполагались одинаковыми, так что память, выделенная одним распределителем, могла быть освобождена другим распределителем того же типа.
  - Изменить ресурсы памяти в ходе выполнения программы было невозможно.
  - Распределители памяти имели ряд ограничений на хранение состояния, например, в качестве регистрационной информации или информации о следующем участке выделяемой памяти.
  - Распределители не обменивались при обмене контейнеров.
- Распределители памяти обрабатывались несогласованно, потому что копирующие конструкторы копировали распределителей, а операция присваивания нет. Таким образом, копирующий конструктор, с одной стороны, и конструктор по умолчанию в сочетании с операцией присваивания, с другой стороны, приводили к разным результатам.

Рассмотрим пример:

```
std::list<int> list1;     // эквивалент:
                          // std::list<int,std::allocator<int>> list1;
std::list<int,MyAlloc<int>> list2;

list1 == list2;           // ОШИБКА: разные типы
list1 = list2;            // ОШИБКА: разные типы
```

Это относится даже к строкам, которые являются разновидностью контейнеров STL.

```
typedef std::basic_string<char, std::char_traits<char>,
                          MyAlloc<char>> MyString;
std::list<std::string> list3;
std::list<MyString,MyAlloc<MyString>> list4;
...
list4.push_back(list3.front());              // ОШИБКА: разные типы элементов
std::equal_range(list3.begin(),list3.end(),
                 list4.begin(),list4.end()); // ОШИБКА: разные типы элементов
```

Для решения этих проблем в стандарте C++11 предложено несколько изменений.

- Компоненты библиотеки больше не должны считать, что все распределители памяти конкретного типа являются одинаковыми.
- *Свойства распределителей памяти* устраняют несколько проблем, создаваемых распределителями памяти, имеющими состояние.
- *Адаптер распределителей памяти с ограниченной областью видимости* предоставляет пользователю возможность передавать распределитель памяти от контейнера к содержащимся в нем элементам.

Эти усовершенствования образуют *инструментарий* для создания полиморфного распределителя памяти, позволяющего сравнивать или присваивать два контейнера с распределителями памяти разных типов, которые являются производными от общего типа распределителей памяти, используемого контейнерами. Однако из-за временных ограничений стандартный полиморфный распределитель памяти не вошел в стандарт C++11.[6]

Таким образом, предназначение класса `scoped_allocator_adaptor` заключается в предоставлении возможности передавать распределитель памяти от контейнера к содержащимся в нем элементам. Тем не менее для обеспечения обратной совместимости стандарт C++11 по-прежнему поддерживает старую модель распределителей памяти по умолчанию. Некоторые распределители памяти и свойства распределителей позволяют переключаться на новую модель, которую класс `scoped_allocator_adaptor<>` использует для обеспечения удобного интерфейса.

Для согласованного использования распределителя памяти контейнером и его элементами можно написать следующие выражения:

```
#include <scoped_allocator>

// используем стандартный распределитель памяти
// для контейнера и всех элементов
std::list<int,std::scoped_allocator_adaptor<std::allocator<int>>> list1;

// используем распределитель памяти MyAlloc<>
// для контейнера и всех элементов
std::list<int,std::scoped_allocator_adaptor<MyAlloc<int>>> list2;
```

Вместо этого можно использовать шаблонные синонимы (см. раздел 3.1.9). Рассмотрим пример:

```
#include <scoped_allocator>

template <typename T, typename Alloc>
using MyList = std::list<T,std::scoped_allocator_adaptor<Alloc<T>>>;

MyList<int,MyAlloc<int>>, list2;
```

Для использования разных распределителей памяти для элементов и контейнера необходимо передать распределитель памяти для элементов как дополнительный шаблонный параметр.

```
std::list<int,std::scoped_allocator_adaptor<MyAlloc1<int>, MyAlloc2<int>>> list;
```

Точно так же можно передать дополнительные типы распределителей памяти, чтобы задать распределитель памяти для элементов и т.д.

S.3.2. Пользовательские распределители памяти в стандарте C++

Как было указано в разделе 19.2, можно просто создать собственный распределитель памяти, создав следующие компоненты.

- Определение типа для `value_type`, который передается как шаблонный параметр.
- Шаблонный конструктор, копирующий внутреннее состояние при изменении типа.

[6] Благодарю Пабло Гальперна (Pablo Halpern) за советы при написании этого раздела.

- Функция-член `allocate()`, выделяющая новую память.
- Функция-член `deallocate()`, освобождающая память, которая больше не нужна.
- Конструктор и деструктор, если они нужны, которые инициализируют, копируют и очищают внутреннее состояние.
- Операции == и !=.

Однако до принятия стандарта C++11 многие значения по умолчанию для пользовательских распределителей памяти не поддерживались. В результате приходилось создавать дополнительные члены, более или менее сложные. Рассмотрим аналог распределителя, описанного в разделе 19.2, который соответствует стандарту C++98:

```
// alloc/myalloc03.hpp

#include <cstddef>
#include <memory>
#include <limits>

template <typename T>
class MyAlloc {
   public:
      // определения типов
      typedef std::size_t    size_type;
      typedef std::ptrdiff_t difference_type;
      typedef T*             pointer;
      typedef const T*       const_pointer;
      typedef T&             reference;
      typedef const T&       const_reference;
      typedef T              value_type;

      // конструкторы и деструктор
      // - ничего не делают, потому что распределитель
      // памяти не имеет состояния
      MyAlloc() throw() {
      }
      MyAlloc(const MyAlloc&) throw() {
      }

      template <typename U>
      MyAlloc (const MyAlloc<U>&) throw() {
      }
      ~MyAlloc() throw() {
      }

      // выделяем память, но не инициализируем num элементов типа T
      T* allocate (std::size_t num, const void* hint = 0) {
          // выделяем память глобальной операцией new
          return static_cast<T*>(::operator new(num*sizeof(T)));
      }

      // освобождаем область с адресом p, в которой записаны удаленные элементы
      void deallocate (T* p, std::size_t num) {
         // освобождаем область памяти с помощью глобальной операции delete
```

```
        ::operator delete(p);
    }

    // возвращаем адрес значений
    T* address (T& value) const {
        return &value;
    }

    const T* address (const T& value) const {
        return &value;
    }

    // возвращаем максимальное количество элементов,
    // которые можно разместить в памяти
    std::size_t max_size () const throw() {
        // см. numeric_limits в разделе 5.3
        return std::numeric_limits<std::size_t>::max() / sizeof(T);
    }

    // инициализируем элементы выделенной памяти с адресом p
    // значением value
    void construct (T* p, const T& value) {
        // инициализируем память с помощью операции new с размещением
        ::new((void*)p)T(value);
    }

    // уничтожаем элементы, содержащиеся в инициализированной
    // области памяти с адресом p
    void destroy (T* p) {
        // уничтожаем объекты, вызывая их деструктор
        p->~T();
    }

    // связываем распределитель памяти с типом U
    template <typename U>
    struct rebind {
        typedef MyAlloc<U> other;
    };
};

// возвращаем признак того, что все специализации
// данного распределителя эквивалентны
template <typename T1, typename T2>
bool operator== (const MyAlloc<T1>&,
                 const MyAlloc<T2>&) throw() {
    return true;
}

template <typename T1, typename T2>
bool operator!= (const MyAlloc<T1>&,
                 const MyAlloc<T2>&) throw() {
    return false;
}
```

Как видим, нам пришлось создать несколько дополнительных типов, функции-члены address(), max_size(), construct(), destroy() и структуру rebind. Подробное описание этих членов приведено в разделе S.3.4.

Шаблонный член rebind<>

Частью распределителя памяти является структура rebind<>. Эта шаблонная структура позволяет любому распределителю памяти выделять память для другого типа косвенным образом. Например, если Allocator — это тип распределителя, то Allocator::rebind<T2>::other — это распределитель, специализированный для элементов типа T2.

Структуру rebind<> можно использовать при реализации контейнера, в котором требуется выделить память для объекта, имеющего тип, отличающийся от типа элементов контейнера. Например, для реализации очереди обычно требуется память для массивов, управляющих блоками элементов (см. типичную реализацию класса deque в разделе 7.4). Таким образом, нам требуется распределитель памяти для выделения массивов указателей на элементы.

```
namespace std {
   template <typename T,
             typename Allocator = allocator<T> >
   class deque {
      ...
   private:
      // связываем распределитель памяти с типом T*
      typedef typename allocator_traits<Allocator>::rebind_alloc
              PtrAllocator;
      Allocator     alloc;       // распределитель памяти для значений типа T
      PtrAllocator  block_alloc; // распределитель памяти для значений типа T*
      T**           elems;       // массив блоков элементов
     ...
   };
}
```

Для управления элементами очереди необходимо иметь один распределитель памяти для блоков элементов и другой для массива блоков элементов. Второй распределитель памяти имеет тип PtrAllocator, совпадающий с типом распределителя памяти для элементов. Используя структуру rebind<> распределитель памяти для элементов (Allocator) связывается с типом массива элементов (T*).

До принятия стандарта C++11 тип PtrAllocator должен был определяться следующим образом:

```
typedef typename Allocator:: template rebind<T*>::other
        PtrAllocator;
```

S.3.3. Распределитель памяти по умолчанию

Распределитель памяти по умолчанию, который используется в тех случаях, когда среди шаблонных параметров не указан никакой конкретный распределитель памяти, объявлен следующим образом:[7]

[7] Благодарю Пабло Гальперна (Pablo Halpern) за советы при написании этого раздела.

```
namespace std {
   template <typename T>
   class allocator {
      public:
         // определения типов
         typedef size_t      size_type;
         typedef ptrdiff_t   difference_type;
         typedef T*          pointer;
         typedef const T*    const_pointer;
         typedef T&          reference;
         typedef const T&    const_reference;
         typedef T           value_type;

     // связываем распределитель памяти с типом U
     template <typename U>
     struct rebind {
          typedef allocator<U> other;
     };

     // возвращаем адрес значений
     pointer address(reference value) const noexcept;
     const_pointer address(const_reference value) const noexcept;

     // конструкторы и деструктор
     allocator() noexcept;
     allocator(const allocator&) noexcept;
     template <typename U>
     allocator(const allocator<U>&) noexcept;
     ~allocator();

     // возвращаем максимальное количество элементов,
     // которое можно разместить в памяти
     size_type max_size() const noexcept;

   // выделяем, но не инициализируем num элементов типа T
    pointer allocate(size_type num,
                     allocator<void>::const_pointer hint = 0);

   // инициализируем элементы, размещенные в области с адресом p,
   // значениями из списка аргументов args
   template <typename U, typename... Args>
   void construct(U* p, Args&&... args);

   // удаляем элементы инициализированной области памяти с адресом p
   template <typename U>
     void destroy(U* p);

   // освобождаем область памяти с адресом p,
   // в которой были записаны удаленные элементы
   void deallocate(pointer p, size_type num);
   };
}
```

Для выделения и освобождения памяти распределитель памяти по умолчанию использует глобальные операции `new` и `delete`. Таким образом, функция-член `allocate()` может генерировать исключение `bad_alloc`. Однако распределитель памяти по умолчанию может быть оптимизирован с помощью повторного использования освобожденной памяти или выделения большего количества памяти, чем требуется, чтобы сэкономить время на дополнительных операциях выделения памяти. Итак, точные моменты времени, в которые выполняются операции `new` и `delete`, не определены. В разделе 19.2 приведена возможная реализация распределителя памяти по умолчанию.

Распределитель памяти по умолчанию имеет следующую специализацию для типа `void`:

```
namespace std {
   template <>
   class allocator<void> {
      public:
         typedef void* pointer;
         typedef const void* const_pointer;
         typedef void value_type;
         template <typename U>
        struct rebind {
          typedef allocator<U> other;
      };
   };
}
```

S.3.4. Подробное описание распределителей

Для того чтобы выполнить указанные требования, распределители памяти должны иметь следующие типы и операции. Распределители, не предназначенные для стандартных контейнеров, могут иметь меньшее количество компонентов.

Определения типов

***allocator*::value_type**

- Тип элементов.
- Обычно эквивалентен `T` для `allocator<T>`.

***allocator*::size_type**

- Тип целочисленных значений без знака, представляющих размер самого большого объекта в модели выделения памяти.
- По умолчанию это тип `std::make_unsigned<allocator::difference_type>::type`, который обычно эквивалентен типу `std::size_t`.

***allocator*::difference_type**

- Тип целочисленных значений со знаком, который представляет разность между двумя указателями в модели выделения памяти.
- По умолчанию это тип `std::pointer_traits<allocator::pointer>::difference_type`, который обычно эквивалентен типу `std::ptrdiff_t`.

***allocator*::pointer**

- Тип указателя на тип элемента.
- По умолчанию: `value_type*`.

***allocator*::const_pointer**

- Тип указателя на тип элемента.
- По умолчанию: `const value_type*`.

***allocator*::void_pointer**

- Тип указателя на тип `void`.
- По умолчанию: `void*`.
- Начиная со стандарта C++11.

***allocator*::const_void_pointer**

- Тип константного указателя на тип `void`.
- По умолчанию: `const void*`.
- Начиная со стандарта C++11.

***allocator*::reference**

- Тип ссылки на тип элемента.
- Обычно эквивалентен типу `T&` для `allocator<T>`.
- После принятия стандарта C++11 больше не требуется.

***allocator*::const_reference**

- Тип константной ссылки на тип элемента.
- Обычно эквивалентен типу `const T&` для `allocator<T>`.
- После принятия стандарта C++11 больше не требуется.

***allocator*::rebind**

- Шаблонная структура, обеспечивающая любому распределителю памяти возможность косвенным образом выделять память для другого типа
- Должна быть объявлена следующим образом (это объявление по умолчанию принято в стандарте C++11).

```
template <typename T>
   class allocator {
      public:
         template <typename U>
         struct rebind {
            typedef allocator<U> other;
         };
         ...
   }
```

- Объяснение предназначения структуры `rebind<>` приведено в разделе S.3.2.

***allocator*::propagate_on_container_copy_assignment**

- Тип свойств для сигнализации о том, что распределитель памяти должен копироваться при выполнении операции присваивающего копирования контейнера.
- По умолчанию: `false_type`.
- Начиная со стандарта C++11.

***allocator*::propagate_on_container_move_assignment**

- Тип свойств для сигнализации о том, что распределитель памяти должен перемещаться при выполнении операции перемещающего копирования контейнера.
- По умолчанию: `false_type`.
- Начиная со стандарта C++11.

***allocator*::propagate_on_container_swap**

- Тип свойств для сигнализации о том, что распределитель памяти должен обмениваться при обмене контейнеров.
- По умолчанию: `false_type`.
- Начиная со стандарта C++11.

Операции

***allocator*::allocator ()**

- Конструктор по умолчанию
- Создает объект распределителя памяти.
- После принятия стандарта C++11 больше не требуется (однако хотя бы один некопирующий/неконвертирующий конструктор должен существовать).

***allocator*::allocator (const allocator& a)**

- Копирующий конструктор.
- Копирует объект распределителя памяти так, что область памяти, выделенная оригиналом или копией, может быть удалена одним из них.

***allocator*::allocator (allocator&& a)**

- Перемещающий конструктор.
- Если он не предусмотрен. используется копирующий конструктор.
- Перемещает объект распределителя памяти так, чтобы область, выделенная объектом `a`, могла быть освобождена объектом `*this`.
- Начиная со стандарта C++11.

***allocator*::~allocator ()**

- Деструктор.
- Уничтожает объект распределителя памяти.

`pointer` ***allocator*`::`address** `(reference` *value*`)`
`const_pointer` ***allocator*`::`address** `(const_reference` *value*`)`

- Первая версия возвращает неконстантный указатель на неконстантное значение *value*.
- Вторая форма возвращает константный указатель на константу *value*.
- После принятия стандарта C++11 больше не требуется.

`size_type` ***allocator*`::`max_size** `()`

- Возвращает наибольшее разумное значение, которое можно передать функции-члену `allocate()` для выделения памяти.
- По умолчанию: `std::numeric_limits<allocator::size_type>::max()`.

`pointer` ***allocator*`::`allocate** `(size_type` *num*`)`
`pointer` ***allocator*`::`allocate** `(size_type` *num*`,`
`allocator<void>::const_pointer` *hint*`)`

- Обе версии возвращают память для *num* элементов типа `T`.
- Элементы не создаются/инициализируются (конструкторы не вызываются).
- Смысл необязательного второго элемента зависит от реализации. Например, его можно использовать для повышения производительности программы.
- Если *num* равно `0`, то возвращаемое значение не определено.

`void` ***allocator*`::`deallocate** `(pointer` *p*`, size_type` *num*`)`

- Освобождает память, на которую ссылается указатель *p*.
- Область памяти, на которую ссылается указатель *p*, должна быть выделена функцией-членом `allocate()` того же самого или эквивалентного распределителя памяти.
- Элементы должны быть предварительно уничтожены.

`void` ***allocator*`::`construct** `(U*` *p*`, Args&&...` *args*`)`

- Инициализирует память для одного элемента, на который ссылается указатель *p*, списком аргументов *args*.
- По умолчанию: `::new((void*)`*p*`) U(std::forward<Args>(`*args*`)...)`.
- До принятия стандарта второй аргумент имел тип `const T&`.

`void` ***allocator*`::`destroy** `(U*` *p*`)`

- Уничтожает объект, на который ссылается указатель *p*, без освобождения памяти.
- Вызывает деструктор объекта.
- По умолчанию: `p->~U()`.

`bool` **operator** `==` `(const` *allocator*`&` *a1*`, const` *allocator*`&` *a2*`)`

- Возвращает `true`, если распределители памяти *a1* и *a2* взаимозаменяемые.
- Два распределителя памяти являются взаимозаменяемыми, если область памяти, выделенная ими, может быть освобождена любым из них.

- До принятия стандарта C++11 при реализации стандартных контейнеров распределители памяти одного и того же типа должны были быть взаимозаменяемыми, поэтому эта функция всегда возвращала значение `true`.

`bool` **operator** `!=` (`const` *allocator*`&` *a1*, `const` *allocator*`&` *a2*)

- Возвращает значение `true`, если два распределителя памяти не являются взаимозаменяемыми.
- Эквивалент `!` (*a1* `==` *a2*).
- До принятия стандарта C++11 при реализации стандартных контейнеров распределители памяти одного и того же типа должны были быть взаимозаменяемыми, поэтому эта функция всегда возвращала значение `false`.

allocator **select_on_container_copy_construction** ()

- Возвращает распределитель памяти, допускающий копирование при создании копии контейнера.
- По умолчанию возвращает `*this`.
- Начиная со стандарта C++11.

S.3.5. Подробное описание утилит для работы с неинициализированной памятью

В разделе описываются вспомогательные функции для работы с неинициализированной памятью. Реализация этих функций, безопасная с точки зрения исключений, публикуется с разрешения Грега Колвина (Greg Colvin).

`void` **uninitialized_fill** (`ForwardIterator` *beg*, `ForwardIterator` *end*, `const T&` *value*)

- Инициализирует элементы в интервале [*beg*,*end*) значением *value*.
- Функция либо выполняется успешно, либо не выполняется вообще.
- Функция обычно реализуется следующим образом:

```
namespace std {
   template <typename ForwIter, typename T>
   void uninitialized_fill(ForwIter beg, ForwIter end,
                           const T& value)
   {
      typedef typename iterator_traits<ForwIter>::value_type VT;
              ForwIter save(beg);
      try {
          for (; beg!=end; ++beg) {
               ::new (static_cast<void*>(&*beg))VT(value);
          }
      }
      catch (...) {
         for (; save!=beg; ++save) {
           save->~VT();
         }
```

```
            throw;
        }
    }
}
```

```
ForwardIterator uninitialized_fill_n (ForwardIterator beg,
Size num, const T& value)
```

- Инициализирует *num* элементов, начиная с позиции *beg*, значением *value*.
- Возвращает позицию, следующую за последним инициализированным элементом.
- Функция либо выполняется успешно, либо не выполняется вообще.
- Функция обычно реализуется следующим образом:

```
namespace std {
   template <typename ForwIter, typename Size, typename T>
   ForwIter uninitialized_fill_n (ForwIter beg, Size num,
                                  const T& value)
   {
      typedef typename iterator_traits<ForwIter>::value_type VT;
              ForwIter save(beg);
      try {
         for ( ; num--; ++beg) {
              ::new (static_cast<void*>(&*beg))VT(value);
         }
         return beg;
      }
      catch (...) {
         for (; save!=beg; ++save) {
              save->~VT();
         }
         throw;
      }
   }
}
```

- Пример использования функции `uninitialized_fill_n()` приведен в разделе 19.3.
- До принятия стандарта C++11 возвращала значение типа `void`.

```
ForwardIterator uninitialized_copy (InputIterator sourceBeg,
                                    InputIterator sourceEnd,
                                    ForwardIterator destBeg)
```

- Инициализирует память, начиная с позиции *destBeg*, элементами из интервала [*sourceBeg*,*sourceEnd*).
- Возвращает позицию, следующую за последним инициализированным элементом.
- Функция либо выполняется успешно, либо не выполняется вообще.
- Функция обычно реализуется следующим образом:

```
namespace std {
   template <typename InputIter, typename ForwIter>
```

```
ForwIter uninitialized_copy(InputIter pos, InputIter end,
                            ForwIter dest)
{
   typedef typename iterator_traits<ForwIter>::value_type VT;
           ForwIter save(dest);
   try {
       for (; pos!=end; ++pos,++dest) {
            ::new (static_cast<void*>(&*dest))VT(*pos);
       }
       return dest;
   }
   catch (...) {
       for (; save!=dest; ++save) {
            save->~VT();
       }
       throw;
   }
}
```

- Пример использования функции `uninitialized_copy()` приведен в разделе 19.3.

`ForwardIterator` **uninitialized_copy_n** `(InputIterator` *sourceBeg*`, Size` *num*`, ForwardIterator` *destBeg*`)`

- Инициализирует *num* элементов памяти, начиная с позиции *destBeg*, элементами, начиная с позиции *sourceBeg*.
- Возвращает позицию, следующую за последним инициализированным элементом.
- Функция либо выполняется успешно, либо не выполняется вообще.
- Функция обычно реализуется следующим образом:

```
namespace std {
   template <typename ForwIter, typename Size, typename ForwIter>
   ForwIter uninitialized_copy_n(InputIter pos, Size num,
                                 ForwIter dest)
   {
      typedef typename iterator_traits<ForwIter>::value_type VT;
              ForwIter save(dest);
      try {
          for ( ; num>0; ++pos,++dest,--num) {
               ::new (static_cast<void*>(&*dest))VT(*pos);
          }
          return dest;
      }
      catch (...) {
          for (; save!=dest; ++save) {
               save->~VT();
          }
          throw;
      }
   }
```

- Начиная со стандарта C++11.

Библиография

В библиографии перечислены источники, упомянутые, позаимствованные или процитированные в книге, а также многочисленные электронные форумы по программированию, существовавшие на момент издания книги.

Веб-сайты обычно изменяются намного быстрее, чем книги и статьи. Интернет-ссылки, перечисленные здесь, в будущем могут стать недействительными, поэтому я веду список источников, которые считаю стабильными, на своем веб-сайте (`http://www.cppstdlib.com`).

Новостные группы и форумы

В Интернете существует множество форумов и тематических конференций, посвященных программированию, языку C++, стандартной библиотеке C++ и библиотеке STL. Некоторые из них *модерируются*, что заметно улучшает их качество, потому что каждое выступление проверяется на приемлемость.

Ниже приведен наиболее важные тематические конференции и форумы.

- Тематическая конференция **`comp.lang.c++.moderated`**, модерируемый форум для технических дискуссий о языке C++ (см. `http://groups.google.com/group/comp.lang.c++.moderated/about`)
- Тематическая конференция **`comp.std.c++`**, модерируемый форум для обсуждения стандарта C++ (см. `http://groups.google.com/group/comp.std.c++/about`)
- Модерируемый форум **`stackoverflow.com`** для дискуссий по программированию, на котором можно использовать дескрипторы, например `c++` или `c++11`, чтобы принять участие в обсуждении специальных тем, посвященных языку C++ (см. `http://stackoverflow.com/tags/c++/info`)
- Тематическая конференция **`comp.lang.c++`**, немодерируемый форум для дискуссий о языке C++ (см. `http://groups.google.com/group/comp.lang.c++/about`)
- Тематическая конференция **`alt.comp.lang.learn.c-c++`**, немодерируемый форум для начинающих программистов на языках C и C++ (см. `http://groups.google.com/group/alt.comp.lang.learn.c-c++/about`)

Книги и веб-сайты

[*Abrahams:RValues*]
Dave Abrahams. ***Move It With Rvalue References***
`http://cpp-next.com/archive/2009/09/move-it-with-rvalue-references/`

[*Austern:STL*]
Matthew H. Austern. ***Generic Programming and the STL: Using and Extending the C++ Standard Template Library***. Reading, MA: Addison-Wesley, 1998

[*Becker:LibExt*]
Pete Becker. ***The C++ Standard Library Extensions: A Tutorial and Reference***. Reading, MA: Addison-Wesley, 2007

[*Becker:RValues*]
Thomas Becker. ***C++ Rvalue References Explained***
`http://thbecker.net/articles/rvalue_references/section_01.html`

[*Boehm:C++MM*]
Hans J. Boehm. ***Threads and memory model for C++***
`http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/`

[*BoehmAdve:MemMod*]
Hans J. Boehm and Sarita V. Adve. ***Foundations of the C++ Concurrency Memory Model***
`http://www.hpl.hp.com/techreports/2008/HPL-2008-56.html`

[*Boost*] ***Boost C++ Libraries*** `http://www.boost.org/`

[*Breymann:STL*] Ulrich Breymann. ***Komponenten entwerfen mit der STL***. Bonn, Germany: Addison-Wesley, 1999 [*C++Std1998*]

ISO. ***Information Technology—Programming Languages—C++***. Document Number ISO/IEC 14882-1998. ISO/IEC, 1998

[*C++Std2003*]
ISO. ***Information Technology—Programming Languages—C++, Second Edition***. Document Number ISO/IEC 14882-2003. ISO/IEC, 2003

[*C++Std2011*]
ISO. ***Information Technology—Programming Languages—C++, Third Edition***. Document Number ISO/IEC 14882-2011. ISO/IEC, 2011

[*C++Std2011Draft*]
Pete Becker, ed. ***Working Draft, Standard for Programming Language C++ (C++11)***
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`

[*ECMAScript*]
Ecma International. ***ECMAScript Language Specification (ECMA-262)***
`http://www.ecma-international.org/publications/standards/Ecma-262.htm`

[*Eggink:C++IO*]
Bernd Eggink. ***Die C++ iostreams-Library***. München, Germany: Hanser Verlag, 1995

[*EllisStroustrup:ARM*]
Margaret A. Ellis and Bjarne Stroustrup. ***The Annotated C++ Reference Manual (ARM)***. Reading, MA: Addison-Wesley, 1990

[*HoadZobel:HashCombine*]
Timothy C. Hoad and Justin Zobel. ***Methods for Identifying Versioned and Plagiarised Documents*** `http://www.cs.rmit.edu.au/~jz/fulltext/jasist-tch.pdf`

[*GlassSchuchert:STL*]
Graham Glass and Brett Schuchert. ***The STL <Primer>***. Englewood Cliffs, NJ: Prentice-Hall, 1996

[*GoF:DesignPatterns*]
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. ***Design Patterns: Elements of Reusable Object-Oriented Software***. Reading, MA: Addison-Wesley, 1994

[*ISO639:LangCodes*]
Codes for the Representation of Names of Languages
`http://www.loc.gov/standards/iso639-2/php/English_list.php`

[*ISO3166:CodeTab*]
ISO 3166-1 decoding table
`http://www.iso.org/iso/support/country_codes/iso_3166_code_lists/iso-3166-1_decoding_table.htm`

[*ISOLatin1*]
ISO/IEC 8859-1
`http://en.wikipedia.org/wiki/ISO_8859-1`

[*ISOLatin9*]
ISO/IEC 8859-15
`http://en.wikipedia.org/wiki/ISO_8859-15`

[*JustThread*]
Anthony Williams. ***C++ Standard Thread Library***
`http://www.stdthread.co.uk/`

[*Karlsson:Boost*]
Bj,orn Karlsson. ***Beyond the C++ Standard Library: An Introduction to Boost***. Reading, MA: Addison-Wesley, 2006

[*KoenigMoo:Accelerated*]
Andrew Koenig and Barbara E. Moo. ***Accelerated C++: Practical Programming by Example***. Boston, MA: Addison-Wesley, 2000

[*Meyers:MoreEffective*]
Scott Meyers. ***More Effective C++: 35 New Ways to Improve Your Programs and Designs***. Reading, MA: Addison-Wesley, 1996

[*Milewski:Atomics*]
Bartosz Milewski. ***C++ atomics and memory ordering***
`http://bartoszmilewski.wordpress.com/2008/12/01`

[*Milewski:Multicore*]
Bartosz Milewski. ***Multicores and Publication Safety***
`http://bartoszmilewski.wordpress.com/2008/08/04`

[*MusserSaini:STL*]
David R. Musser and Atul Saini. ***STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library***. Reading, MA: Addison-Wesley, 1996

[*N1456:HashTable*]
Matthew Austern. ***A Proposal to Add Hash Tables to the Standard Library (revision 4)***
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1456.html

[*N2351:SharedPtr*]
Peter Dimov and Beman Dawes. ***Improving shared_ptr for C++0x, Revision 2***
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2351.htm

[*N2480:MemMod*]
Hans-J. Boehm. ***A Less Formal Explanation of the Proposed C++ Concurrency Memory Model***
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2480.html

[*N2543:FwdList*]
Matt Austern. ***STL singly linked lists (revision 3)***
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2543.htm

[*N2661:Chrono*]
Howard E. Hinnant, Walter E. Brown, Jeff Garland, and Marc Paterno. ***A Foundation to Sleep On: Clocks, Points in Time, and Time Durations***
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2661.html

[*N3051:DeprExcSpec*]
Doug Gregor. ***Deprecating Exception Specifications***
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3051.html

[*N3194:Futures*]
Lawrence Crowl, Anthony Williams, and Howard Hinnant. ***Clarifying C++ Futures***
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3194.htm

[*N3198:DeprAdapt*]
Daniel Kr¸ugler. ***Deprecating unary_function and binary_function (Revision 1)***
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3198.htm

[*N3279:LibNoexcept*]
Alisdair Meredith and John Lakos. ***Conservative use of noexcept in the Library***
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3279.pdf

[*Nelson:C++*]
Mark Nelson. ***C++ Programmer's Guide to the Standard Template Library***. Foster City, CA: IDG Books Worldwide, 1995

[*Plauger:C++Lib*]
P. J. Plauger. ***The Draft Standard C++ Library***. Englewood Cliffs, NJ: Prentice Hall, 1995

[*SafeSTL*]
Cay S. Horstmann. ***Safe STL***
http://www.horstmann.com/safestl.html

[*STLport*]
STLport
http://www.stlport.org/

[*Stroustrup:C++*]
Bjarne Stroustrup. ***The C++ Programming Language, Third Edition***. Reading, MA: Addison-Wesley, 1997

[*Stroustrup:C++0x*]
Bjarne Stroustrup. ***What is C++0x?***
`http://www2.research.att.com/~bs/what-is-2009.pdf`

[*Stroustrup:Design*]
Bjarne Stroustrup. ***The Design and Evolution of C++***. Reading, MA: Addison-Wesley, 1994

[*Stroustrup:FAQ*]
Bjarne Stroustrup. ***C++11 — the recently approved new ISO C++ standard***
`http://www.research.att.com/~bs/C++11FAQ.html`

[*Sutter:LockFree*]
Herb Sutter. ***Lock-Free Code: A False Sense of Security***
`http://drdobbs.com/cpp/210600279`

[*Teale:C++IO*]
Steve Teale. ***C++ IOStreams Handbook***. Reading, MA: Addison-Wesley, 1993

[*UTF8*]
UTF-8
`http://en.wikipedia.org/wiki/UTF-8`

[*VisualC++Locales*]
Visual C++ Language and Country/Region Strings
`http://msdn.microsoft.com/en-us/library/hzz3tw78.aspx`

[*Williams:C++Conc*]
Anthony Williams. ***C++ Concurrency in Action: Practical Multithreading***. Greenwich, CT: Manning, 2012

[*Williams:CondVar*]
Anthony Williams. ***Multithreading and Concurrency: Condition Variable Spurious Wakes***
`http://www.justsoftwaresolutions.co.uk/threading/?page=2`

Предметный указатель

Б

К

Л

М

С

Ц

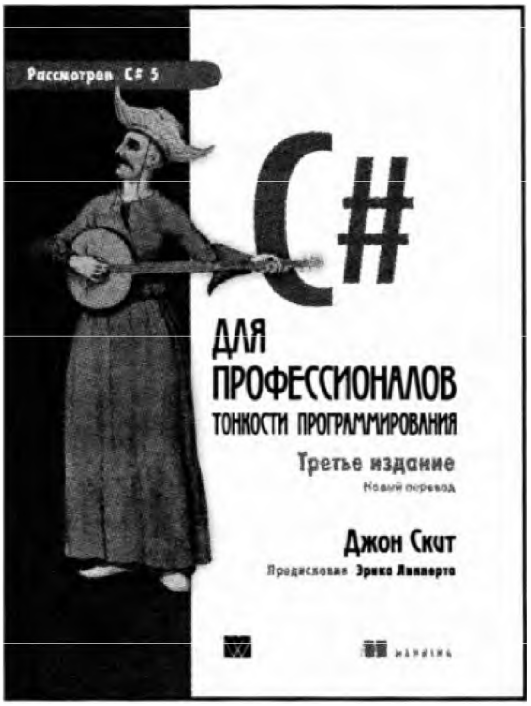
Ч

Ш

C# ДЛЯ ПРОФЕССИОНАЛОВ ТОНКОСТИ ПРОГРАММИРОВАНИЯ

3-Е ИЗДАНИЕ

Джон Скит



www.williamspublishing.com

Если вы занимаетесь разработкой приложений .NET, то будете использовать C# как при построении сложного приложения уровня предприятия, так и при ускоренном написании какого-нибудь чернового приложения. В C# 5 можно делать удивительные вещи с помощью обобщений, лямбда-выражений, динамической типизации, LINQ, итераторных блоков и других средств. Однако прежде их необходимо должным образом изучить. Это издание было полностью пересмотрено с целью раскрытия новых средств версии C# 5, включая тонкости написания сопровождаемого асинхронного кода. Вы увидите всю мощь языка C# в действии и научитесь работать с ценнейшими средствами, которые эффективно впишутся в применяемый набор инструментов. Кроме того, вы узнаете, как избегать скрытых ловушек при программировании на C# с помощью простых и понятных объяснений вопросов, касающихся внутреннего устройства языка.

ISBN 978-5-8459-1909-0

в продаже

MICROSOFT ASP.NET 4.5 С ПРИМЕРАМИ НА C#5.0 ДЛЯ ПРОФЕССИОНАЛОВ

5-е издание

Адам Фримен

www.williamspublishing.com

ASP.NET 4.5 остается превалирующей технологией от Microsoft для построения динамических веб-сайтов, предлагая разработчикам невероятную мощь и гибкость.

Настоящая книга представляет собой наиболее полный справочник по ASP.NET, который только можно найти. Полностью переписанное 5-е издание предлагает все, что необходимо знать для создания качественно спроектированных веб-сайтов ASP.NET. Книга начинается с основных концепций и постепенно формирует все нужные профессиональные навыки. В книге будет показано, как работать с базами данных, рассмотрены многочисленные применения XML и описаны соображения относительно защиты сайта от злоумышленников. Наконец, будут представлены сложные темы, такие как использование проверки достоверности на стороне клиента, jQuery и Ajax.

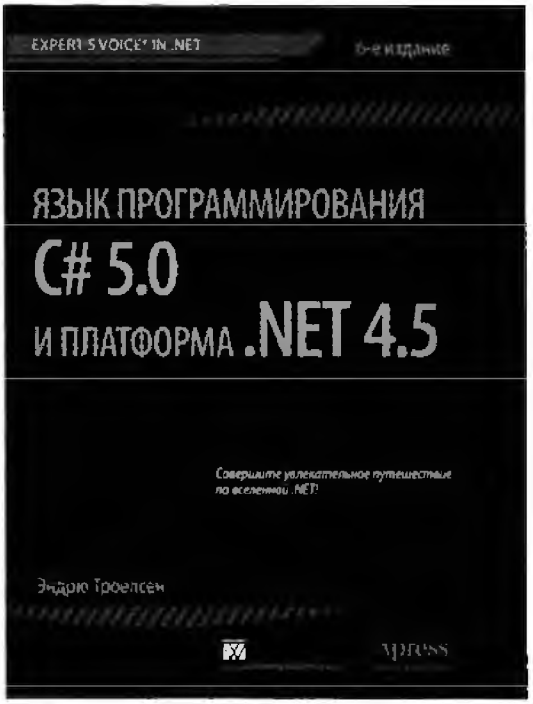
Эта книга рассчитана на разработчиков с базовыми знаниями платформы .NET Framework, которые хотят научиться ее использовать в профессиональной среде.

ISBN 978-5-8459-1878-9

в продаже

ЯЗЫК ПРОГРАММИРОВАНИЯ C# 5.0 И ПЛАТФОРМА .NET 4.5
6-Е ИЗДАНИЕ

Эндрю Троелсен



www.williamspublishing.com

Новое издание этой книги было полностью пересмотрено и переписано с учетом последних изменений в спецификации языка C# и дополнений платформы .NET Framework. Отдельные главы посвящены важным новым средствам, которые превращают .NET Framework 4.5 в самое передовое решение для корпоративных приложений. Помимо этого, рассмотрены все ключевые возможности языка C#, как старые, так и новые, что позволило обрести популярность предыдущим изданиям этой книги (материал покрывает все темы, начиная с обобщений и кончая pLINQ). Основное назначение книги — служить исчерпывающим руководством по языку программирования C# и ключевым аспектам платформы .NET (сборкам, удаленному взаимодействию, Windows Forms, Web Forms, ADO.NET, веб-службам XML и т.д.).

ISBN 978-5-8459-1814-7

в продаже

C# 5.0 И ПЛАТФОРМА .NET 4.5
ДЛЯ ПРОФЕССИОНАЛОВ

К. Нейгел, Б. Ивьен,
Д. Глинн, К. Уотсон,
М. Скиннер

www.dialektika.com

Книга известных специалистов в области разработки приложений с использованием .NET Framework посвящена программированию на языке C# 5.0 в среде .NET Framework 4.5. Ее отличает простой и доступный стиль изложения, изобилие примеров и множество рекомендаций по написанию высококачественных программ. Подробно рассматриваются такие вопросы, как основы языка программирования C#, организация среды .NET, работа с данными, написание Windows- и веб-приложений, взаимодействие через сеть, создание веб-служб и многое другое. Немалое внимание уделено проблемам безопасности и сопровождения кода. Тщательно подобранный материал позволит без труда разобраться с тонкостями использования ASP.NET и построения веб-страниц, а также научиться разрабатывать приложения для Windows 8 и WinRT. Читатели ознакомятся с работой в Visual Studio, а также с применением различных технологий, встроенных в .NET.

Книга рассчитана на программистов разной квалификации, а также будет полезна для студентов и преподавателей дисциплин, связанных с программированием и разработкой для .NET.

ISBN 978-5-8459-1850-5 **в продаже**

C# 5.0
СПРАВОЧНИК
ПОЛНОЕ ОПИСАНИЕ ЯЗЫКА

Джозеф Албахари,
Бен Албахари

Данное руководство ставшее бестселлером, позволяет получить точные ответы практически на любые вопросы по C# 5.0 и .NET CLR. Уникально организованное по концепциям и сценариям использования, обновленное пятое издание книги предлагает реорганизованные разделы, посвященные параллелизму, многопоточности и параллельному программированию, а также включает подробные материалы по новому средству C# 5.0 — асинхронным функциям.
Проверенная более чем 20 экспертами, в числе которых Эрик Липперт, Стивен Тауб, Крис Барроуз и Джон Скит, эта книга содержит все, что необходимо для освоения C# 5.0. Она широко известна как исчерпывающий справочник по языку.

www.williamspublishing.com

ISBN 978-5-8459-1819-2 **в продаже**

ВВЕДЕНИЕ В ИНФОРМАЦИОННЫЙ ПОИСК

Кристофер Д. Маннинг,
Прабхакар Рагхаван,
Хайнрих Шютце

Это первый учебник, в котором наряду с классическим поиском рассматриваются веб-поиск, а также классификация и кластеризация текстов. Учебник написан с точки зрения информатики и содержит современное изложение всех аспектов проектирования и реализации систем сбора, индексирования и поиска документов, методов оценки таких систем, а также введение в методы машинного обучения на базе коллекций текстов.

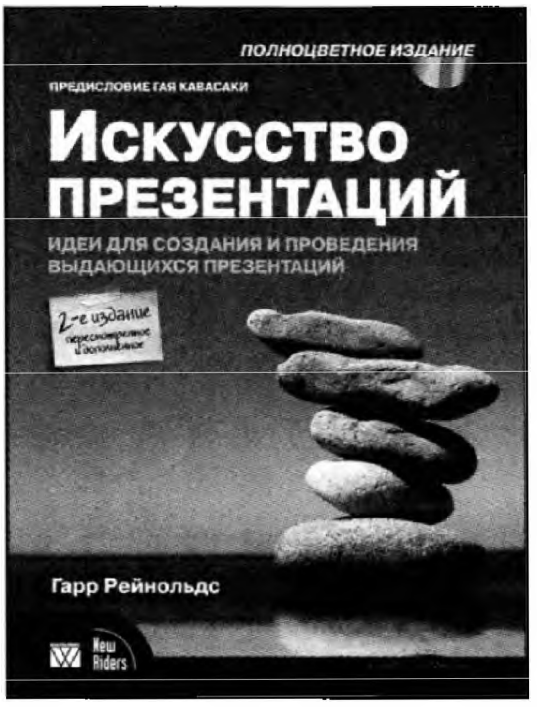
Книга задумана как вводный курс по информационному поиску, но будет интересна исследователям и профессионалам.

www.williamspublishing.com

ISBN 978-5-8459-1623-5 **в продаже**

ИСКУССТВО ПРЕЗЕНТАЦИЙ
ИДЕИ ДЛЯ СОЗДАНИЯ И ПРОВЕДЕНИЯ ВЫДАЮЩИХСЯ ПРЕЗЕНТАЦИЙ
2-Е ИЗДАНИЕ, ПЕРЕСМОТРЕННОЕ И ДОПОЛНЕННОЕ

Гарр Рейнольдс



www.williamspublishing.com

В новое издание своего бестселлера Гарр Рейнольдс включил свежие примеры, из которых читатели могут черпать вдохновение, а также описание методов, позволяющих прорваться сквозь шум и отвлекающие факторы современной жизни и сделать аудиторию активным участником презентации. Сочетая надежные и проверенные принципы разработки презентаций с принципами простоты, исповедуемыми философией дзэн, эта книга выводит на путь создания более простых и в то же время более эффективных презентаций, которые будут по достоинству оценены публикой, надолго останутся в ее памяти и, что самое главное, изменят к лучшему ее жизнь.

ISBN 978-5-8459-1846-8 **в продаже**

БЕСТСЕЛЛЕР ПО C++ ОБНОВЛЕН С УЧЕТОМ СТАНДАРТА C++11

Стандартная библиотека C++ содержит набор универсальных классов и интерфейсов, значительно расширяющих ядро языка C++. Однако эта библиотека не является самоочевидной. Для того чтобы полнее использовать возможности ее компонентов и извлечь из них максимальную пользу, необходим полноценный справочник, а не простое перечисление классов и их функций.

В данной книге описывается библиотека как часть нового стандарта ANSI/ISO C++ (C++11). Здесь содержится исчерпывающее описание каждого компонента библиотеки, включая его предназначение и структуру; очень подробно описываются сложные концепции и тонкости практического программирования, необходимые для их эффективного использования, а также ловушки и подводные камни; приводятся точные сигнатуры и определения наиболее важных классов и функций, а также многочисленные примеры работоспособных программ. Основным предметом изучения книги является стандартная библиотека шаблонов (STL), в частности контейнеры, итераторы, функциональные объекты и алгоритмы.

В книге описаны все новые компоненты библиотеки, вошедшие в стандарт C++11, в частности:

- Параллельная работа
- Арифметика рациональных чисел
- Часы и таймеры
- Кортежи
- Новые контейнеры STL
- Новые алгоритмы STL
- Новые интеллектуальные указатели
- Случайные числа и распределения
- Свойства типов и утилиты
- Регулярные выражения

В книге также рассматриваются новый стиль программирования на C++ и его влияние на стандартную библиотеку, включая лямбда-функции, диапазонные циклы for, семантику перемещения и вариативные шаблоны.

Книге посвящен специальный веб-сайт www.cppstdlib.com, на котором, в частности, можно найти исходные коды программ.

НИКОЛАИ М. ДЖОСАТТИС — независимый технический консультант, разрабатывающий программные системы среднего и крупного масштаба для телекоммуникационных, финансовых и промышленных компаний. Бывший член рабочей группы Комитета по стандартизации C++, широко известный в программистском сообществе как автор популярных книг. Кроме книги *Стандартная библиотека C++*, ставшей мировым бестселлером после ее первой публикации в 1999 году, он является автором книги *C++ Templates: The Complete Guide (Addison-Wesley,* 2003; русский перевод: Вандервурд Д., Джосаттис Н. *Шаблоны C++: справочник разработчика.* — М.: Издательский дом "Вильямс", 2003) и *SOA in Practice: The Art of Distributed System Design* (O'Reilly Media, 2007).

informit.com/aw

cppstdlib.com

ISBN 978-5-8459-1837-6

13027

9 785845 918376

ВИЛЬЯМС

Издательский дом "Вильямс"
www.williamspublishing.com