

ДЛЯ ПРОФЕССИОНАЛОВ

C++

**Стандартная
библиотека**



ДИПТЕР



Стандартная библиотека C++ содержит общие классы и интерфейсы, значительно расширяющие базовый язык C++. В этой книге отражены новейшие компоненты стандартной библиотеки C++, входящие в полный стандарт ANSI/ISO C++. Особое внимание уделяется библиотеке STL (Standard Template Library), ее контейнерам, операторам, объектам функций и алгоритмам. Также подробно рассмотрены специальные контейнеры, строки, численные классы, проблемы интернационализации и потоки ввода/вывода.

Для каждого компонента приводится разностороннее описание — предназначение и строение компонента, примеры, подробное описание, потенциальные трудности и ошибки, а также точные сигнатуры и определения всех классов и их функций.

Здесь вы найдете не только исчерпывающую документацию по всем компонентам библиотеки, но и доступные разъяснения многих сложных понятий, практические советы, необходимые для их эффективного использования, а также многочисленные примеры программ.

Подробная, исчерпывающая, понятная и практичная, эта книга станет незаменимым источником информации о C++, к которому вы будете обращаться снова и снова.

Широта охвата Глубина изложения Профессиональный подход

Уровень пользователя: Опытный/эксперт **Серия:** Для профессионалов

ISBN 5-94723-635-4



9 785947 236354

Посетите наш web-магазин: <http://www.piter.com>


Addison Wesley

5112



9 785947 236354
Джосьюэс Н. С++. Стандартная
библиотека C++
Цена 530.00

Библиосфера

Сертификат № 12. Б-4
Москва, 2000 г.

Nicolai M. Josuttis

The C++ Standard Library

A Tutorial and Reference



Addison-Wesley

004.43
1426

Николай Джосьюотис

для профессионалов

C++

**Стандартная
библиотека**

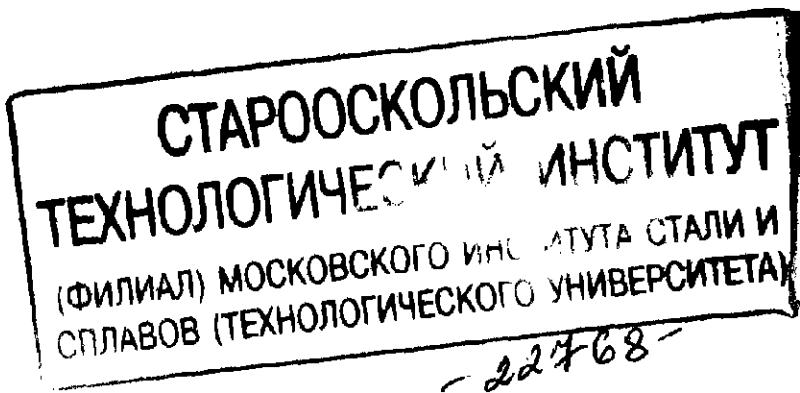


Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2004

ББК 32.973-018.1

УДК 681.3.06

Д42



Д42 С++ Стандартная библиотека. Для профессионалов / Н. Джосьюис. — СПб: Питер, 2004. — 730 с.: ил.

ISBN 5-94723-635-4

Книга посвящена стандартной библиотеке C++, значительно расширяющей возможностей базового языка. Особое внимание уделяется стандартной библиотеке шаблонов STL — многочисленным контейнерам, итераторам, объектам функций и алгоритмам. Также подробно описана библиотека потокового ввода/вывода `IOStream` и другие компоненты стандартной библиотеки C++: специализированные контейнеры, строки, поддержка математических вычислений, проблемы интернационализации и т. д.

Кроме подробной документации по всем классам и интерфейсам библиотеки в книге также разъясняются многие нетривиальные концепции, а также рассматриваются практические аспекты программирования, необходимые для эффективного использования стандартной библиотеки. Типичные ошибки. Материал поясняется многочисленными примерами. Книга может использоваться как учебник, и как справочник. Книга рассчитана на программистов среднего и высокого уровня.

ББК 32.973-018.

УДК 681.3.06

Права на издание получены по соглашению с Addison-Wesley Longman.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 0201379260 (англ.)
ISBN 5-94723-635-4

© 1999 by Addison-Wesley
© Перевод на русский язык ЗАО Издательский дом «Питер», 2004
© Издание на русском языке, оформление ЗАО Издательский дом «Питер», 2004

Краткое содержание

Предисловие	15
Благодарности	16
Глава 1. О книге	18
Глава 2. Знакомство с C++ и стандартной библиотекой	23
Глава 3. Общие концепции	40
Глава 4. Вспомогательные средства	50
Глава 5. Стандартная библиотека шаблонов	86
Глава 6. Контейнеры STL	152
Глава 7. Итераторы STL	257
Глава 8. Объекты функций STL	295
Глава 9. Алгоритмы STL	320
Глава 10. Специальные контейнеры	422
Глава 11. Строки	454
Глава 12. Числовые типы	510
Глава 13. Ввод-вывод с использованием потоковых классов	558
Глава 14. Интернационализация	657
Глава 15. Распределители памяти	701
Ресурсы Интернета	716
Библиография	718
Алфавитный указатель	720

Содержание

Предисловие	.
Благодарности	.
Глава 1. О книге	.
Зачем написана эта книга	.
Что необходимо знать читателю	.
Стиль и структура книги	.
Как читать эту книгу	.
Текущая ситуация	.
От издателя перевода	.
Глава 2. Знакомство с C++ и стандартной библиотекой	.
История	.
Новые языковые возможности	.
Шаблоны	.
Явная инициализация базовых типов	.
Обработка исключений	.
Пространства имен	.
Тип <code>bool</code>	.
Ключевое слово <code>explicit</code>	.
Новые операторы преобразования типа	.
Инициализация константных статических членов класса	.
Определение <code>main</code>	.
Сложность алгоритмов	.
Глава 3. Общие концепции	.
Пространство имен <code>std</code>	.
Заголовочные файлы	.
Обработка ошибок и исключений	.
Стандартные классы исключений	.
Члены классов исключений	.
Генерирование стандартных исключений	.
Классы исключений, производные от <code>exception</code>	.
Распределители памяти	.

Объекты функций	134
Понятие объекта функции	134
Стандартные объекты функций	140
Элементы контейнеров	143
Требования к элементам контейнеров	143
Семантика значений и ссылочная семантика	144
Ошибки и исключения внутри STL	146
Обработка ошибок	146
Обработка исключений	148
Расширение STL	151
Глава 6. Контейнеры STL	152
Общие возможности и операции	152
Общие возможности контейнеров	152
Общие операции над контейнерами	153
Векторы	156
Возможности векторов	157
Операции над векторами	159
Векторы как обычные массивы	164
Обработка исключений	164
Примеры использования векторов	165
Класс <code>vector<bool></code>	167
Деки	169
Возможности деков	170
Операции над деками	171
Обработка исключений	173
Примеры использования деков	173
Списки	174
Возможности списков	175
Операции над списками	176
Обработка исключений	180
Примеры использования списков	181
Множества и мульти множества	183
Возможности множеств и мульти множеств	184
Операции над множествами и мульти множествами	185
Обработка исключений	194
Примеры использования множеств и мульти множеств	194
Пример определения критерия сортировки на стадии выполнения	198
Отображения и мультиотображения	200
Возможности отображений и мультиотображений	202
Операции над отображениями и мультиотображениями	203
Отображения как ассоциативные массивы	212
Обработка исключений	214
Примеры использования отображений и мультиотображений	214
Пример с отображениями, строками и изменением критерия сортировки на стадии выполнения	218

Другие контейнеры STL	221
Строки как контейнеры STL	222
Обычные массивы как контейнеры STL	223
Хэш-таблицы	225
Реализация ссылочной семантики	226
Рекомендации по выбору контейнера	229
Типы и функции контейнеров	233
Определения типов	234
Операции создания, копирования и уничтожения	236
Немодифицирующие операции	237
Присваивание	241
Прямой доступ к элементам	242
Операции получения итераторов	243
Вставка и удаление элементов	245
Специальные функции для списков	250
Поддержка распределителей памяти	253
Обработка исключений в контейнерах STL	254
Глава 7. Итераторы STL	257
Заголовочные файлы итераторов	257
Категории итераторов	257
Итераторы ввода	258
Итераторы вывода	259
Прямые итераторы	260
Двунаправленные итераторы	261
Итераторы произвольного доступа	261
Проблема увеличения и уменьшения итераторов в векторах	264
Вспомогательные функции итераторов	265
Перебор итераторов функцией <code>advance</code>	265
Обработка расстояния между итераторами функцией <code>distance</code>	267
Перестановка элементов функцией <code>iter_swap</code>	269
Итераторные адаптеры	270
Обратные итераторы	270
Итераторы вставки	275
Потоковые итераторы	281
Трактовка итераторов	287
Написание унифицированных функций для итераторов	289
Пользовательские итераторы	291
Глава 8. Объекты функций STL	295
Концепция объектов функций	295
Объект функции в качестве критерия сортировки	296
Объекты функций с внутренним состоянием	298
Возвращаемое значение алгоритма <code>for_each</code>	301
Предикаты и объекты функций	303

Стандартные объекты функций	305
Функциональные адаптеры	306
Функциональные адаптеры для функций классов	307
Функциональные адаптеры для обычных функций	310
Написание пользовательских объектов функций для функциональных адаптеров	311
Дополнительные композиционные адаптеры	313
Унарные композиционные адаптеры	314
Бинарные композиционные адаптеры	318
Глава 9. Алгоритмы STL	320
Заголовочные файлы алгоритмов	320
Общий обзор алгоритмов	321
Введение	321
Классификация алгоритмов	322
Вспомогательные функции	332
Алгоритм <code>for_each</code>	333
Немодифицирующие алгоритмы	336
Подсчет элементов	336
Минимум и максимум	338
Поиск элементов	340
Сравнение интервалов	352
Модифицирующие алгоритмы	358
Копирование элементов	358
Преобразование и объединение элементов	361
Обмен интервалов	364
Присваивание	366
Замена элементов	368
Алгоритмы удаления	371
Удаление элементов с заданным значением	371
Удаление дубликатов	374
Перестановочные алгоритмы	378
Перестановка элементов в обратном порядке	379
Циклический сдвиг элементов	380
Перестановка элементов	383
Перестановка элементов в случайном порядке	384
Перемещение элементов в начало	387
Алгоритмы сортировки	388
Сортировка всех элементов	388
Частичная сортировка	391
Разбиение по n-му элементу	394
Сортировка в куче	396
Алгоритмы упорядоченных интервалов	399
Поиск элементов	400
Слияние интервалов	405

Численные алгоритмы	414
Обработка интервалов	414
Преобразования относительных и абсолютных значений	417
Глава 10. Специальные контейнеры	422
Стеки	422
Основной интерфейс	423
Пример использования стека	424
Строение класса <code>stack</code>	425
Пользовательская реализация стека	427
Очереди	430
Основной интерфейс	431
Пример использования очереди	432
Строение класса <code>queue</code>	433
Пользовательская реализация очереди	435
Приоритетные очереди	438
Основной интерфейс	439
Пример использования приоритетных очередей	440
Строение класса <code>priority_queue</code>	441
Битовые поля	444
Примеры использования битовых полей	444
Строение класса <code>bitset</code>	447
Глава 11. Строки	454
Общие сведения	454
Пример построения имени временного файла	455
Пример чтения слов и вывода символов в обратном порядке	459
Описание строковых классов	462
Строчные типы	462
Операции со строками	463
Конструкторы и деструкторы	466
Строки и C-строки	466
Размер и емкость	468
Обращение к символам	469
Операции сравнения	471
Модификация строк	472
Подстроки и конкатенация	474
Операторы ввода-вывода	475
Поиск	476
Значение <code>pros</code>	478
Поддержка итераторов для строк	480
Интернационализация	485
Эффективность	488
Строки и векторы	488
Строение строковых классов	489
Определения типов и статические значения	489
Операции создания, копирования и уничтожения строк	490

Операции с размером и емкостью	492
Операции сравнения	493
Обращение к символам	494
Построение С-строк и символьных массивов	495
Модифицирующие операции	496
Поиск	502
Выделение подстрок и конкатенация	505
Функции ввода-вывода	506
Получение итераторов	507
Поддержка распределителей памяти	508
Глава 12. Числовые типы	510
Комплексные числа	510
Примеры использования класса <code>complex</code>	511
Операции с комплексными числами	513
Строение класса <code>complex</code>	520
Массивы значений	525
Знакомство с массивами значений	525
Подмножества элементов в массивах значений	531
Строение класса <code>valarray</code>	545
Классы подмножеств элементов	551
Глобальные математические функции	556
Глава 13. Ввод-вывод с использованием потоковых классов	558
Общие сведения о потоках ввода-вывода	559
Потоковые объекты	559
Потоковые классы	559
Глобальные потоковые объекты	560
Потоковые операторы	560
Манипуляторы	561
Простой пример	562
Основные потоковые классы и объекты	562
Иерархия потоковых классов	562
Глобальные потоковые объекты	566
Заголовочные файлы	567
Стандартные операторы <code><< и >></code>	567
Оператор вывода <code><<</code>	568
Оператор ввода <code>>></code>	569
Ввод-вывод специальных типов	570
Состояние потока данных	571
Константы состояния потока данных	571
Функции для работы с состоянием потока данных	573
Состояние потока данных и логические условия	574
Состояние потока данных и исключения	576
Стандартные функции ввода-вывода	581
Функции ввода	581
Функции вывода	585
Пример использования	586

Манипуляторы	586
Принципы работы манипуляторов	587
Пользовательские манипуляторы	589
Форматирование	589
Форматные флаги	590
Форматированный ввод-вывод логических данных	592
Ширина поля, заполнитель, выравнивание	593
Отображение знака для положительных чисел и вывод в верхнем регистре	595
Система счисления	596
Формат вещественных чисел	598
Общие параметры форматирования	600
Интернационализация	601
Доступ к файлам	602
Режимы открытия файлов	606
Произвольный доступ к файлам	609
Файловые дескрипторы	611
Связывание потоков ввода-вывода	612
Нежесткое связывание функцией tie	612
Жесткое связывание с использованием потоковых буферов	613
Перенаправление стандартных потоков данных	615
Потоки чтения и записи	617
Потоковые классы для работы со строками	619
Классы строковых потоков данных	619
Потоковые классы char*	623
Операторы ввода-вывода для пользовательских типов	625
Реализация операторов вывода	626
Реализация операторов ввода	627
Ввод-вывод с использованием вспомогательных функций	630
Пользовательские операторы с функциями неформатированного ввода-вывода	631
Пользовательские форматные флаги	633
Правила построения пользовательских операторов ввода-вывода	635
Классы потоковых буферов	636
Потоковые буфера с точки зрения пользователя	636
Итераторы потоковых буферов	638
Пользовательские потоковые буфера	641
Проблемы эффективности	653
Синхронизация со стандартными потоками данных С	653
Буферизация в потоковых буферах	654
Непосредственная работа с потоковыми буферами	655
Глава 14. Интернационализация	657
Различия в кодировках символов	658
Расширенные и многобайтовые кодировки	658
Трактовки символов	659
Интернационализация специальных символов	663

Концепция локального контекста	664
Использование локальных контекстов	665
Фацеты	670
Строение объекта локального контекста	672
Строение фацетов	676
Числовое форматирование	677
Форматирование времени и даты	681
Форматирование денежных величин	684
Классификация и преобразования символов	689
Контекстная сортировка	697
Интернационализация сообщений	699
Глава 15. Распределители памяти	701
Использование распределителей в прикладном программировании	701
Использование распределителей при программировании библиотек	702
Инициализирующий итератор	705
Временные буферы	705
Распределитель по умолчанию	706
Пользовательский распределитель	708
Требования к распределителям памяти	710
Определения типов	710
Операции	711
Операции с неинициализированной памятью	713
Ресурсы Интернета	716
Группы Usenet	716
URL-адреса	716
Библиография	718
Алфавитный указатель	720

Предисловие

Сначала я намеревался написать небольшую книгу на немецком языке (около 400 страниц) о стандартной библиотеке C++. Это было в 1993 году. Теперь, десять лет спустя, вы видите результат — книга содержит более 700 страниц, заполненных фактическим материалом, рисунками и примерами. Я стремился описать стандартную библиотеку C++ так, чтобы читатель нашел ответы на все (или почти все) вопросы еще до того, как они у него возникнут. Впрочем, данная книга не претендует на полное описание всех аспектов стандартной библиотеки C++. Вместо этого я постарался представить основные аспекты изучения и программирования на C++ в контексте стандартной библиотеки.

Все темы излагаются по одному общему принципу: от теоретического материала я перехожу к конкретным подробностям, необходимым для решения повседневных задач программирования. Примеры программ помогут читателю лучше освоить представленный материал.

Вот и все. Надеюсь, чтение этой книги принесет вам столько же удовольствия, сколько мне приносило ее написание.

Благодарности

Идеи, концепции, решения и примеры, представленные в книге, были позаимствованы из многих источников. В каком-то смысле даже несправедливо, что на обложке стоит только мое имя, поэтому я хочу поблагодарить всех людей и организаций, помогавшие мне на протяжении последних лет.

Прежде всего, я благодарен Дитмару Кюлю (Dietmar Kühl), эксперту по C++ (особенно в области потоков данных и интернационализации — он реализовал библиотеку потоков данных просто для развлечения). Дитмар не только перевел большие фрагменты книги с немецкого на английский, но и написал некоторые части книги, руководствуясь собственным опытом. Кроме того, все эти годы он оказывал мне неоценимую помощь, делясь своим мнением по поводу материалов книги.

Далее я хочу поблагодарить всех рецензентов и читателей, поделившихся своим мнением (список получился довольно длинным, поэтому я прошу прощать за возможные упущения). Рецензентами английской версии книги были Чак Эллисон (Chuck Allison), Грет Комо (Greg Comeau), Джеймс А. Кротингер (James A. Crottinger), Гэбриел Дос Рейс (Gabriel Dos Reis), Аллан Эзаст (Alan Ezust), Натан Майерс (Nathan Myers), Вернер Моснер (Werner Mossner), Todd Вельдхуизен (Todd Weldhuizen), Чичианг Ван (Chichiang Wan), Джуди Вард (Judy Ward) и Томас Викхульт (Thomas Wikhult). Немецкую версию рецензировали Ральф Бекер (Ralph Boecker), Дирк Херманн (Dirk Herrmann), Дитмар Кюль (Dietmar Kühl), Эdda Лорке (Edda Lörke), Герберт Шубнер (Herbert Scheubner), Доминик Штрассер (Dominik Strasser) и Мартин Вейцель (Martin Weitzel). Кроме того, дополнительную информацию предоставили Мэтт Остерн (Matt Austern), Валентин Боннард (Valentin Bonnard), Грет Колвин (Greg Kolvin), Беман Доус (Beman Dawes), Билл Гиббоис (Bill Gibbons), Лоис Голдтвейт (Lois Goldthwaite), Эндрю Кениг (Andrew Koenig), Стив Рамсби (Steve Rumsby), Бъярн Страуструп (Bjarne Stroustrup) и Дэвид Вандеворд (David Vandevoorde).

Я особенно благодарен Дэйву Абрахамсу (Dave Abrahams), Дженет Кокер (Janet Cocker), Кэтрин Охала (Catherine Ohala) и Морин Уиллард (Maureen Willard) за внимательное изучение и редактирование всей книги. Их отзывы внесли неоценимый вклад в улучшение ее качества.

Хочу особо выделить Херба Саттера (Herb Sutter), автора знаменитой рассылки «*Guru of the Week*» (серии задач по программированию C++, регулярно публикуемой в группе новостей `comp.lang.c++.moderated`).

Я также благодарен всем людям и организациям, которые помогали мне в тестировании примеров на разных платформах и с разными компиляторами. Спасибо

Стиву Адамчику (Steve Adamczyk), Майку Андерсону (Mike Anderson) и Джону Спайсеру (John Spicer) из EDG за превосходный компилятор и техническую поддержку. Их работа способствовала процессу стандартизации и написанию 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 Metrowerks за пробную версию среды программирования CodeWarrior. Огромное спасибо разработчикам бесплатных компиляторов GNU и egcs. Спасибо компании Microsoft за пробную версию Visual C++. Я благодарен Роланду Хартингеру (Roland Hartinger) из Siemens Nixdorf Information Systems AG за пробную версию компилятора C++. Спасибо Topjects GmbH за пробную версию библиотеки ObjectSpace.

Большое спасибо сотрудникам издательства Addison Wesley Longman, работавшим со мной. В частности, 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 Рабиновиц (Marty Rabinowitz), Сьюзен Спицеर (Susan Spitzer) и Морин Уиллард (Maureen Willard). С вами было интересно работать.

Также я благодарен сотрудникам BREDEX GmbH и всем участникам сообщества C++ (и особенно участникам процесса стандартизации) за поддержку и терпение при ответах на мои особенно глупые вопросы.

Остается лишь сердечно поблагодарить мою семью: Улли, Лукаса, Анику и Фредерика. Боюсь, я не уделял им должного внимания во время работы над книгой.

Спасибо всем!

1 О книге

Зачем написана эта книга

Вскоре после своего появления язык C++ превратился в фактический стандарт объектно-ориентированного программирования. Отсюда логично возникла потребность в стандартизации. Только при наличии общепринятого стандарта можно написать программу, работающую на разных платформах, от РС до больших машин. Кроме того, при наличии стандартных *библиотек* программист использует универсальные компоненты и без потери переносимости программ работает на более высоком уровне абстракции, а не пишет весь код с самого начала.

Процесс стандартизации C++ был начат в 1989 году международным комитетом ANSI/ISO. Комитет разработал стандарт, основанный на знаменитых книгах Бъярна Страуструпа «The C++ Programming Language» и «The Annotated C++ Reference Manual». В 1997 году работа над стандартом была завершена, а в 1998 году он был принят в качестве международного стандарта ISO и ANSI. Процесс стандартизации включал разработку стандартной библиотеки C++, расширяющей базовые возможности языка и содержащей ряд компонентов общего назначения. Используя средства определения новых абстрактных и обобщенных типов, библиотека предоставляет в распоряжение программиста набор общих классов и интерфейсов. В частности, стандартная библиотека содержит:

- строковые типы;
- различные структуры данных (например, динамические массивы, связанные списки и бинарные деревья);
- различные алгоритмы (например, алгоритмы сортировки);
- классы для представления числовых данных;
- классы ввода-вывода;
- классы, обеспечивающие интернационализацию программ.

Все перечисленные возможности доступны через относительно простой программный интерфейс. Стандартные компоненты чрезвычайно важны для многих программ. В наши дни обработка данных обычно сопряжена с вводом, обработкой и выводом больших объемов данных, которые часто представляются в текстовом формате.

Стандартную библиотеку нельзя назвать простой и понятной. Чтобы работать с ее компонентами и пользоваться их преимуществами, недостаточно простого перечисления классов и их функций — требуется хорошее объяснение основных концепций и важных подробностей. Именно с этой целью и была написана эта книга. Сначала читатель знакомится с библиотекой и всеми ее компонентами на концептуальном уровне, а затем получает более подробную информацию, необходимую для практического программирования. Принципы использования всех компонентов поясняются конкретными примерами. Таким образом, книга представляет собой подробное введение в библиотеку C++ как для новичков, так и для более опытных программистов. Вооружившись новыми знаниями, вы сможете в полной мере использовать возможности стандартной библиотеки C++.

Необходимо отметить, что далеко не весь материал книги прост и очевиден. Стандартная библиотека весьма гибка, однако в любых нетривиальных ситуациях гибкость не дается даром. Учтите, что использование библиотеки иногда сопряжено со всевозможными каверзами и подвохами; на них указывается по мере изложения материала, а также предлагаются возможные способы обойти проблемы.

Что необходимо знать читателю

Предполагается, что читатель уже владеет языком C++ (в книге описаны стандартные компоненты C++, но не сам язык). Он должен быть знаком с концепциями классов, наследования, шаблонов и обработки ошибок. Тем не менее книга не требует досконального знания всех тонкостей языка. Действительно важные подробности описаны в книге, а мелочи существенны в основном для программистов, занимающихся реализацией библиотеки, а не ее использованием. Следует помнить, что язык изменился в процессе стандартизации, поэтому некоторые ваши знания могут оказаться устаревшими. На с. 25 приведен краткий обзор основных средств языка, необходимых для использования библиотеки. Прочтайте этот материал, если вы еще недостаточно хорошо освоили все новые средства C++ (например, ключевое слово `template` и концепцию пространств имен).

Стиль и структура книги

Многие компоненты стандартной библиотеки C++ в той или иной степени зависят друг от друга, поэтому трудно описывать отдельные компоненты без упоминания остальных частей. Отсюда вытекают несколько возможных вариантов подачи материала. Например, можно было бы приводить описания в порядке их следования в стандарте C++. Однако просто описывать компоненты библиотеки C++ с самого начала — не лучшее решение. Также можно было бы начать с обзора всех компонентов, за которым расположить главы с подробными описаниями. Наконец, компоненты можно было бы отсортировать и попытаться выстроить их в порядке, обеспечивающем минимальное количество перекрестных ссылок. В итоге автор остановился на варианте, в котором сочетаются все три решения. Книга начинается с краткого изложения базовых концепций и вспомогательных

средств библиотеки. Далее следуют описания основных компонентов, каждое из которых занимает одну или несколько глав. На первом месте стоит стандартная библиотека шаблонов (STL). Бессспорно, STL является самой мощной, самой сложной и самой интересной частью библиотеки C++, а ее архитектура в значительной степени влияет на другие компоненты. Затем рассматриваются более очевидные компоненты — специализированные контейнеры, строковые и числовые классы. Вероятно, следующий компонент — библиотека `IOStream` — уже знаком читателю и используется им в практической работе. В завершение будут рассмотрены проблемы интернационализации, которые оказывают некоторое влияние на работу библиотеки `IOStream`.

Знакомство с каждым компонентом начинается с описания целей и архитектурных решений, сопровождаемых примерами. Далее следует углубленное описание различных способов использования компонента и возникающих при этом проблем. Описание, как правило, заканчивается справочным разделом, в котором приводятся сигнатуры и определения классов компонента и их функций.

Ниже дано краткое содержание книги. В первых четырех главах представлена вводная информация о книге и стандартной библиотеке C++ в целом.

- Глава 1, «О книге». В этой главе (которую вы читаете в настоящий момент) представлены тема книги и ее содержание.
- Глава 2, «Знакомство с C++ и стандартной библиотекой». Краткий обзор истории стандартной библиотеки C++ и контекста ее стандартизации. Кроме того, в этой главе приводится общая информация по теме книги, в том числе описания новых языковых средств и концепции сложности.
- Глава 3, «Общие концепции». Базовые принципы строения библиотеки, понимание которых необходимо для использования ее компонентов. В частности, в этой главе представлено пространство имен `std`, формат заголовочных файлов и общие средства обработки ошибок и исключений.
- Глава 4, «Вспомогательные средства». Здесь рассматриваются некоторые вспомогательные средства, предназначенные для пользователей библиотеки и для самой библиотеки. В частности, описаны вспомогательные функции `max()`, `min()` и `swap()`, типы `pair` и `auto_ptr`, а также тип `numeric_limits`, представляющий дополнительную информацию о числовых типах данных в зависимости от реализации.

В главах с 5-й по 9-ю описаны различные аспекты STL.

- Глава 5, «Стандартная библиотека шаблонов». В этой главе представлены общие концепции стандартной библиотеки шаблонов (STL) — сборника контейнеров и алгоритмов, используемых для обработки наборов данных. Обстоятельно изложены основные принципы работы, проблемы и специальные приемы программирования, а также роли основных частей библиотеки.
- Глава 6, «Контейнеры STL». Глава посвящена контейнерным классам STL. Сначала мы рассмотрим различия между векторами, деками, списками, простыми и ассоциативными множествами, затем проанализируем сходство между ними; материал поясняется типичными примерами использования контейнеров. Глава завершается перечислением всех функций контейнеров в виде удобного справочника.

- Глава 7, «Итераторы STL». Подробное описание классов итераторов STL. В частности, рассматриваются разные категории итераторов, вспомогательные функции итераторов и адаптеры (потоковые итераторы, обратные итераторы и итераторы с возможностью вставки).
 - Глава 8, «Объекты функций STL». Описание объектов функций STL.
 - Глава 9, «Алгоритмы STL». Перечисление и описание алгоритмов STL. После краткого вступления и сравнения алгоритмов приводятся подробные описания алгоритмов, сопровождаемые примерами программ.

Главы с 10-й по 12-ю посвящены «простым» стандартным классам.
 - Глава 10, «Специальные контейнеры». Описание специальных контейнерных классов стандартной библиотеки C++. В частности, здесь рассматриваются адаптеры для очередей и стеков, а также класс `bitset`, предназначенный для управления битовыми полями произвольной разрядности.
 - Глава 11, «Строки». В этой главе представлены строковые типы стандартной библиотеки C++ (да, таких типов несколько!). В стандарте C++ строки представлены как «очевидные» фундаментальные типы данных с возможностью использования разнообразных символов.
 - Глава 12, «Числовые типы». Глава посвящена числовым компонентам стандартной библиотеки C++. В частности, в ней описаны типы комплексных чисел и классы для представления массивов числовых значений (применяются при работе с матрицами, векторами и уравнениями).

В главах 13 и 14 рассматриваются темы, относящиеся к вводу-выводу и интернационализации (эти две темы тесно связаны друг с другом).
 - Глава 13, «Ввод-вывод с использованием потоковых классов». Описание подсистемы ввода-вывода C++ – стандартизированной формы известной библиотеки `iostream`. В данной главе также приводятся некоторые важные подробности, которые нередко упускают из виду. Например, здесь рассматривается правильная методика определения и интеграции специальных каналов ввода-вывода, которые на практике часто реализуются неправильно.
 - Глава 14, «Интернационализация». Глава посвящена основным принципам и классам, используемым при интернационализации программ. В частности, в ней рассматриваются проблемы разных кодировок и применение различных форматов при выводе вещественных чисел и дат.
- Оставшуюся часть книги составили глава 15, список ссылок на ресурсы Интернета, библиография и алфавитный указатель.
- Глава 15, «Распределители памяти». Концепции моделей памяти в стандартной библиотеке C++.
 - Список ссылок на ресурсы Интернета с дополнительной информацией по темам, рассмотренным в книге.
 - Список книг и иных источников, которые упоминались, использовались или цитировались в этой книге.
 - Алфавитный указатель.

Как читать эту книгу

Книга представляет собой гибрид учебника и структурированного справочника по стандартной библиотеке C++. Отдельные компоненты стандартной библиотеки C++ в той или иной степени независимы друг от друга, поэтому после глав 2–4 описания компонентов можно читать в любом порядке. Не забывайте, что главы 5–9 посвящены одному и тому же компоненту. Чтобы понять материал остальных глав о STL, начните с вводного описания STL в главе 5.

Программист C++, желающий освоить основные принципы и все аспекты стандартной библиотеки, может просто читать эту книгу от начала до конца, пропуская справочные разделы. При работе с некоторыми компонентами стандартной библиотеки C++ нужную информацию проще всего найти по алфавитному указателю, который сделан достаточно подробным, чтобы свести к минимуму время поиска.

Как известно, все новое лучше всего изучается на конкретных примерах. По этой причине материал книги поясняется многочисленными примерами, от нескольких строк кода до целых программ. В последнем случае имя файла с программой указывается в первой строке комментария. Файлы примеров можно загрузить с web-сайта оригинального издания этой книги (<http://www.josuttis.com/libbook>) или с web-сайта издательства «Питер» (<http://www.piter.com/download>).

Текущая ситуация

На момент написания книги стандарт C++ был полностью подготовлен. Некоторые компиляторы еще не соответствуют новому стандарту. Вероятно, ситуация изменится в ближайшем будущем, однако нельзя исключать того, что часть материала книги не будет соответствовать некоторым системам, и читателю придется вносить изменения в примеры под специфику конкретной платформы. Практически все примеры программ нормально компилируются в EGCS версии 2.8 и выше; реализации этого компилятора для многих платформ бесплатно распространяются через Интернет (<http://egcs.cygnus.com/>) и на компакт-дисках со сборниками программ.

От издателя перевода

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы сможете найти по адресу <http://www.piter.com/download>.

Подробную информацию о наших книгах вы найдете на web-сайте издательства <http://www.piter.com>.

2 Знакомство с C++ и стандартной библиотекой

История

Процесс стандартизации C++ начался в 1989 году и продолжался до конца 1997 года, хотя некоторые формальные предложения отложили его окончательную публикацию до сентября 1998 года. Результатом этой работы стало справочное руководство, насчитывающее около 750 страниц и опубликованное международной организацией по стандартизации (ISO). Стандарт получил название «Information Technology – Programming Languages – C++», ему был присвоен номер ISO/IEC 14882–1998, и он распространяется национальными представительствами ISO (в США это ANSI).

Стандарт стал важной вехой на пути развития C++. Строгие, формализованные определения синтаксиса и правил поведения C++ упрощают преподавание языка, написание программ C++, а также их адаптацию для других платформ. В результате пользователь получает дополнительную свободу выбора между реализациями C++. От повышения надежности и переносимости программ выигрывают разработчики реализаций библиотеки и дополнительного инструментария. Благодаря стандарту прикладные программисты C++ работают быстрее и эффективнее, кроме того, сокращаются затраты времени и сил на их сопровождение.

Одной из составляющих стандарта является стандартная библиотека. Она содержит базовые компоненты для выполнения операций ввода-вывода, для работы со строками, контейнерами (структурами данных) и алгоритмами (сортировка, поиск, слияние и т. д.), для математических вычислений и, как и следовало ожидать от международного стандарта, для поддержки интернационализации, в том числе разных кодировок.

Возможно, вас интересует, почему процесс стандартизации занял почти десять лет. Более того, после близкого знакомства со стандартом возникает другой вопрос — почему после столь долгой работы результат получился не идеальным. Десяти лет оказалось недостаточно! Впрочем, с точки зрения истории и контекста стандартизации сделано довольно много. Результат вполне пригоден для практического применения, хотя и не совершенен (как и все в этом мире).

Дело в том, что стандарт создавался не какой-нибудь компанией с излишками средств и свободного времени. Люди, занимающиеся стандартизацией, за свою работу получают очень мало или не получают вообще ничего. Следовательно, если специалист не работает на компанию, заинтересованную в появлении стандарта, он работает просто для собственного удовольствия. К счастью, в мире еще встречаются энтузиасты, у которых хватает свободного времени и денег.

Стандарт C++ появился не на пустом месте. За основу был взят язык C++ в том виде, в котором он был описан Бъярном Страуструпом, создателем C++. Однако у специализированной библиотеки не было прототипа в виде книги или существующей библиотеки. В библиотеку были включены готовые реализации классов¹. Из-за этого библиотека получилась недостаточно однородной. В разных компонентах применяются разные архитектурные принципы. В качестве примера можно привести различия между строковыми классами и STL, набором структур данных и алгоритмов.

- Проектировщики строковых классов стремились сделать их как можно более надежными и удобными. По этой причине интерфейс строковых классов почти не требует пояснений и выявляет многие ошибки в параметрах.
- Основной целью проектирования STL было объединение разных структур данных с разными алгоритмами при обеспечении оптимального быстродействия. В результате удобство работы с STL оставляет желать лучшего, а многие логические ошибки не проверяются реализациями. Чтобы извлечь максимум пользы из широты возможностей и замечательного быстродействия STL, необходимо знать основные принципы и внимательно применять их на практике.

Все эти компоненты являются частью одной библиотеки. Проектировщики постарались хотя бы отчасти согласовать их, но в целом каждый компонент сохранил свою базовую философию проектирования.

Один из компонентов библиотеки считался фактическим стандартом еще до начала официальной стандартизации; это библиотека `Iostream`. Разработанная в 1984 году, она была реализована заново с частичной переработкой архитектуры в 1989 году. Библиотека уже использовалась во многих программах, поэтому ее общая концепция была сохранена для обеспечения обратной совместимости.

Итоговый стандарт (язык и библиотека) стал результатом обсуждения и участия многих сотен людей со всего мира. Например, японцы внесли ряд полезных предложений из области интернационализации. В процессе обсуждения совершались ошибки, а участники нередко расходились во мнениях. Всем казалось, что в 1994 году стандарт уже близится к завершению, однако включение STL кардинально изменило всю библиотеку. Но чтобы проект все же когда-нибудь завершился, разработчики постепенно перестали принимать новые предложения, какими бы полезными они ни были. Из-за этого в стандарт не была включена такая распространенная структура данных, как хэш-таблица.

Появление стандарта не означает, что эволюция C++ завершена. В стандарте будут исправляться ошибки и иесоответствия; вполне вероятно, что лет через пять появится его следующая версия. Тем не менее на ближайшие годы у про-

¹ Возможно, вас интересует, почему в процессе стандартизации библиотека не была спроектирована заново? Дело в том, что основной целью стандартизации является не изобретение или разработка чего-либо, а согласование уже готовых решений.

граммистов C++ имеется стандарт, а вместе с ним — и возможность написания универсального кода, переносимого на разные платформы.

Новые языковые возможности

Стандартизация базовых возможностей языка и библиотеки C++ происходила параллельно. При таком подходе в библиотеке могли применяться усовершенствованные средства языка, а язык выигрывал от опыта реализации библиотеки. Более того, в процессе стандартизации библиотек часто использовались специальные языковые конструкции, которые еще не поддерживались существующими компиляторами.

Современный язык C++ заметно отличается от C++ пятилетней давности. Если вы не следили за его развитием, возможно, некоторые новые конструкции, используемые в библиотеке, окажутся для вас неожиданностью. В этом разделе приводится краткий обзор новых возможностей языка. За подробностями обращайтесь к специализированным учебникам.

На момент написания книги новые языковые конструкции не поддерживались некоторыми компиляторами. Автор надеется (и не без оснований), что ситуация скоро изменится, тем более что многие разработчики компиляторов участвовали в процессе стандартизации. Тем не менее отдельные случаи несовместимости могут ограничивать использование библиотеки. В переносимых версиях библиотеки обычно учитывается, какие возможности реализованы в конкретной среде (обычно для определения поддерживаемых возможностей запускаются тестовые программы, а затем по результатам проверки настраиваются препроцессорные директивы). Наиболее характерные и важные ограничения будут упоминаться в сносках.

В следующих подразделах рассматриваются основные языковые возможности, важные для стандартной библиотеки C++.

Шаблоны

Практически все составляющие стандартной библиотеки оформлены в виде шаблонов. Без поддержки шаблонов использовать стандартную библиотеку невозможно. Более того, для функционирования библиотеки была нужна поддержка некоторых специальных свойств шаблонов, о которых будет рассказано после краткого обзора.

Шаблонами называются функции или классы, определяемые без указания одного или нескольких типов. При использовании шаблона эти типы передаются в виде аргументов, явно или косвенно. Рассмотрим типичный пример — функцию, которая возвращает большее из двух значений:

```
template <class T>
inline const T& max (const T& a, const T& b)
{
    // Если a<b, вернуть b; иначе вернуть a.
    return a < b ? b : a;
}
```

В первой строке Т определяется как произвольный тип данных, который должен передаваться при вызове функции. В качестве имени параметра может использоваться произвольный идентификатор, но имя Т встречается настолько часто, что почти стало стандартным обозначением. Хотя перед формальным параметром шаблона стоит ключевое слово `class`, он вовсе не обязан быть классом. Допускается любой тип данных, который поддерживает операции, используемые шаблоном¹.

По тому же принципу классы «параметризуются» для произвольных типов. Параметризация особенно удобна для контейнерных классов, поскольку она позволяет определять операции с элементами контейнера для произвольного типа элементов. Стандартная библиотека C++ содержит многочисленные шаблоны контейнеров (за примерами обращайтесь к главам 6 и 10). Впрочем, шаблоны используются и для других целей. Например, строковые классы параметризуются по типу символов и свойствам кодировки (см. главу 11).

Не стоит полагать, что шаблон компилируется только один раз, а в результате компиляции строится код, подходящий для любого типа. В действительности шаблон компилируется для каждого типа или комбинации типов, с которыми он используется. Отсюда следует важная проблема, связанная с практическим применением шаблонов: реализация шаблонной функции должна быть доступна к моменту вызова, чтобы функцию можно было откомпилировать для конкретного типа. Следовательно, в настоящее время единственный переносимый способ использования шаблонов основан на их определении в заголовочных файлах с применением подставляемых (`inline`) функций².

Полноценная реализация стандартной библиотеки C++ должна поддерживать не только шаблоны, но и многие стандартизованные свойства шаблонов. Некоторые из этих свойств рассматриваются ниже.

Нетипизированные параметры шаблонов

Кроме параметров-типов в шаблонах допускается использование нетипизированных параметров. В этом случае нетипизованный параметр считается частью определения типа. Например, аргумент шаблона стандартного класса `bitset<>` (см. с. 444) содержит количество битов. В следующем фрагменте определяются два битовых поля, состоящих из 32 и 50 бит:

```
bitset<32> flags32; // Битовое поле из 32 бит
bitset<50> flags50; // Битовое поле из 50 бит
```

Эти битовые поля относятся к разным типам, потому что при их определении задаются разные аргументы шаблонов. Следовательно, их нельзя присваивать

¹ Ключевое слово `class` было выбрано для того, чтобы избежать введения нового ключевого слова при использовании шаблонов. Тем не менее в современной версии C++ появилось новое ключевое слово `typename`, которое также может использоваться при объявлении шаблонов (см. с. 27).

² Для решения проблемы с обязательным определением шаблонов в заголовочных файлах в стандарт была включена модель компиляции шаблонов с ключевым словом `export`. Впрочем, на момент написания книги она еще нигде не была реализована.

вать или сравнивать друг с другом (при отсутствии соответствующего преобразования типа).

Параметры шаблонов по умолчанию

Шаблоны классов могут иметь параметры по умолчанию. Например, следующий фрагмент разрешает объявлять объекты класса `MyClass` как с одним, так и с двумя аргументами¹:

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

При передаче одного аргумента вместо второго используется параметр по умолчанию:

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

Аргументы шаблонов по умолчанию могут определяться на основе предыдущих аргументов.

Ключевое слово `typename`

Ключевое слово `typename` означает, что следующий за ним идентификатор обозначает тип. Рассмотрим следующий пример:

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

В этом фрагменте ключевое слово `typename` поясняет, что `SubType` является подтипов класса `T`. Следовательно, `ptr` — указатель на тип `T::SubType`. Без `typename` идентификатор `SubType` интерпретируется как статический член класса, а следующая строка воспринимается как умножение значения `SubType` типа `T` на `ptr`:

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

Поскольку идентификатор `SubType` объявлен типом, любой тип, используемый вместо `T`, должен содержать внутренний тип `SubType`. Например, использование типа `Q` в качестве аргумента шаблона, как показано ниже, возможно только при условии, что в `Q` определяется внутренний тип `SubType`:

```
MyClass<Q> x;
```

Объявление выглядит примерно так:

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

¹ Обратите внимание на пробел между символами `>`. Последовательность `>` воспринимается компилятором как оператор сдвига, что приводит к синтаксической ошибке.

В этом случае переменная `ptr` класса `MyClass<Q>` является указателем на тип `int`. Однако подтип также может быть абстрактным типом данных (например, классом):

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

Ключевое слово `typename` всегда должно квалифицировать идентификатор шаблона как тип, даже если другая интерпретация не имеет смысла. В C++ любой идентификатор шаблона, указанный без ключевого слова `typename`, интерпретируется как значение.

Ключевое слово `typename` также может использоваться вместо слова `class` в объявлении шаблона:

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

Шаблонные функции классов

Допускается определение функций классов в виде шаблонов. С другой стороны, такие функции не могут быть виртуальными и не имеют параметров по умолчанию. Пример:

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

Шаблон `MyClass:f` объявляет набор функций с параметром произвольного типа. При вызове функции может передаваться любой аргумент (при условии, что его тип поддерживает все операции, используемые в `f`).

Данная возможность часто используется для автоматического преобразования типов в классах шаблонов. Например, в следующем определении аргумент `x` функции `assign()` должен точно соответствовать типу объекта, для которого он вызывается:

```
template <class T>
class MyClass {
private:
    T value;
public:
    void assign ( const MyClass<T>& x) { // Тип x должен
                                                // соответствовать типу *this
        value = x.value;
    }
    ...
}:
```

Однако использование шаблона с другим типом при вызове `assign()` является ошибкой даже в том случае, если между типами выполняется автоматическое преобразование:

```

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

    d.assign(d); // OK
    d.assign(i); // ОШИБКА: i относится к типу MyClass<int>,
                  // но в данном случае ОБЯЗАТЕЛЕН тип MyClass<double>
}

```

Определение другого шаблона для функции позволяет обойти требование точного совпадения типов. В этом случае аргумент может иметь произвольный тип шаблона (при условии, что типы совместимы по присваиванию):

```

template <class T>
class MyClass {
private:
    T value;
public:
    template <class X> // Шаблон функции позволяет
    void assign (const MyClass<X>& x) { // использовать другие типы
        value = x.getValue(); // шаблонов при присваивании.
    }
    T getValue () const {
        return value;
    }
    ...
};

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

    d.assign(d); // OK
    d.assign(i); // OK (тип int совместим с double по присваиванию)
}

```

Поскольку тип аргумента `x` функции `assign` теперь отличен от `*this`, прямой доступ к закрытым и защищенным членам `MyClass<>` невозможен. Вместо этого приходится использовать обходные пути вроде функции `getValue()` из рассмотренного примера.

Шаблонный конструктор представляет собой особую разновидность шаблона, определяемую внутри класса. Шаблонные конструкторы обычно определяются для обеспечения неявных преобразований типов при копировании объектов. Учтите, что шаблонный конструктор не замещает стандартный копирующий конструктор. При точном соответствии типов будет генерирован и вызван стандартный копирующий конструктор. Пример:

```

template <class T>
class MyClass {
public:
    // Шаблонный конструктор с автоматическим преобразованием типа

```

```

// не замещает стандартный копирующий конструктор
template <class U>
MyClass (const MyClass<U>& x);

};

void f()
{
    MyClass<double> xd;
    ...
    MyClass<double> xd2(xd); // Вызывает стандартный копирующий конструктор
    MyClass<int> xi(xd);    // Вызывает шаблонный конструктор
    ...
}

```

В приведенном примере тип `xd2` соответствует типу `xd`, поэтому объект инициализируется копирующим конструктором по умолчанию. С другой стороны, тип `xi` отличается от типа `xd`, поэтому объект инициализируется шаблонным конструктором. Следовательно, если вы определяете шаблонный конструктор, не забудьте также определить собственный копирующий конструктор, если стандартный копирующий конструктор вас не устраивает. Другой пример шаблона, определяемого как член класса, приведен на с. 50.

Шаблоны вложенных классов

Вложенные классы тоже могут оформляться в виде шаблонов:

```

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

```

Явная инициализация базовых типов

При явном вызове конструктора без аргументов базовые типы инициализируются нулями:

```

int i1;          // Неопределенное значение
int i2 = int(); // Переменная инициализируется нулем

```

Такая возможность была реализована для того, чтобы вы могли написать код шаблона, в котором величине любого типа заведомо будет присвоено некоторое значение по умолчанию. Например, в следующей функции вызов конструктора гарантирует, что для всех базовых типов переменная `x` будет инициализирована нулем:

```

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

```

Обработка исключений

В стандартной библиотеке C++ применяется механизм обработки исключений. С его помощью можно обрабатывать экстренные ситуации, не «загромождая» интерфейс функции лишними аргументами или возвращаемыми значениями. При возникновении внештатной ситуации обычная последовательность обработки данных прерывается «генерированием исключения»:

```
class Error;

void f()
{
    ...
    if (условие) {
        throw Error(); // Создать объект класса Error и передать
                       // в нем информацию об исключении
    }
    ...
}
```

Команда `throw` запускает процесс, называемый *раскруткой стека*. Иначе говоря, программа выходит из всех блоков или функций так, как обычно происходит при выполнении команды `return`, однако при этом управление никуда не передается. Для всех локальных объектов, объявленных в блоках, из которых выходит программа, вызываются деструкторы. Раскрутка стека продолжается до тех пор, пока не произойдет выход из `main()` с завершением программы либо исключение не будет «перехвачено» и обработано секцией `catch`:

```
int main()
{
    try {
        ...
        f();
        ...
    }
    catch (const Error&) {
        ... // Обработка исключения
    }
    ...
}
```

В этом примере любое исключение типа `Error`, возникшее в блоке `try`, будет перехвачено секцией `catch`¹.

Объекты исключений представляют собой стандартные объекты, определяемые в виде обычных классов или базовых типов. Следовательно, вы можете переда-

¹ Исключение завершает вызов функции, в которой оно произошло. При этом стороне, вызвавшей функцию, можно вернуть объект в виде аргумента. Тем не менее, возврат управления не может рассматриваться как своего рода «обратный вызов» (снизу, где была обнаружена проблема, наверх, где эта проблема решается). После обработки исключения управление нельзя вернуть обратно и продолжить выполнение функции с того места, где оно было прервано. В этом отношении обработка исключений принципиально отличается от обработки сигналов.

вать целые типы, строки или шаблоны, являющиеся частью иерархии классов. Обычно в программе создается некое подобие иерархии специальных классов представляющих разные ошибки. В переменных этих объектов можно передавать любую нужную информацию от места обнаружения ошибки к месту ее обработки.

Обратите внимание: механизм называется *обработкой исключений*, а не *обработкой ошибок*. Эти два понятия не всегда эквивалентны. Например, ошибки в данных, введенных пользователем, нередко рассматриваются не как исключение, а как вполне типичное явление. Вероятно, в таких случаях ошибки пользовательского ввода стоит обработать локально с использованием традиционных средств обработки ошибок.

Чтобы указать, какие исключения могут генерироваться функцией, включите в ее объявление *спецификацию исключений*:

```
void f() throw(bad_alloc); // f() генерирует только исключения bad_alloc
```

Пустой список означает, что функция вообще не может создавать исключения:

```
void f() throw(); // f() не создает исключения
```

При нарушении спецификации исключений программа выполняет специальные действия (см. описание класса `bad_exception` на с. 44).

В стандартную библиотеку C++ включены некоторые общие средства обработки исключений, в том числе стандартные классы исключений и класс `auto_ptr` (см. с. 43 и 54).

Пространства имен

С появлением новых библиотек, модулей и компонентов возрастает вероятность конфликтов имен при их одновременном использовании. Пространства имен помогают решить эту проблему.

Пространство имен объединяет идентификаторы в именованной области видимости. При этом имя пространства является единственным глобальным идентификатором, который может конфликтовать с другими глобальными именами. По аналогии с членами классов идентификаторы, входящие в пространство имен, уточняются именем этого пространства с оператором `::`, как показано ниже:

```
// Определение идентификаторов в пространстве имен josuttis
namespace josuttis {
    class File;
    void myGlobalFunc();
    ...
}

// Использование идентификатора, входящего в пространство имен
josuttis::File obj;
...
josuttis::myGlobalFunc();
```

В отличие от классов пространства имен открыты для новых определений и расширений. Следовательно, пространства имен могут использоваться для определения модулей, библиотек или компонентов, причем даже в разных файлах. Пространства имен определяют логические, а не физические модули (в UML и других аналогичных средствах модули также называются *пакетами*).

Если какие-либо типы аргументов определены в пространстве имен, в котором определена функция, то при вызове функции уточнять пространство имен не обязательно (так называемое правило Кенига). Пример:

```
// Определение идентификаторов в пространстве имен josuttis
class File;
void myGlobalFunc( const File&);

}

...
josuttis::File obj;
...
myGlobalFunc(obj); // По правилу Кенига функция josuttis::myGlobalFunc
                   // будет успешно найдена
```

Объявление using помогает избежать утомительных многократных повторений уточняющего префикса. Например, следующее объявление означает, что в текущей области видимости идентификатор *File* является локальным синонимом для *josuttis::File*:

```
using josuttis::File;
```

С другой стороны, *директива using* открывает доступ ко всем именам заданного пространства имен так, как если бы они были объявлены за его пределами. Это может привести к возникновению обычных конфликтов. Например, представленная ниже директива делает идентификаторы *File* и *myGlobalFunc()* глобальными в текущей области видимости:

```
using namespace josuttis;
```

Если в глобальной области видимости уже существуют идентификаторы *File* или *myGlobalFunc()*, то при использовании имен из пространства *josuttis* без уточняющих префиксов компилятор сообщает о неоднозначности ссылки.

Никогда не используйте директиву *using* при отсутствии полной информации о контексте (например, в заголовочных файлах, модулях или библиотеках). Директива может изменить область видимости идентификаторов в пространстве имен, что приведет к нарушению работы программы при включении или использовании программного кода другого модуля. Вообще говоря, присутствие директив *using* в заголовочных файлах считается признаком плохого проектирования.

В стандартной библиотеке C++ все идентификаторы определяются в пространстве имен *std*. За дополнительной информацией обращайтесь на с. 40.

Тип `bool`

Для упрощения работы с логическими (булевыми) переменными в стандартную библиотеку включен тип `bool`. Использование типа `bool` делает программу более наглядной и позволяет переопределять поведение логических величин. Логические значения представляются литералами `true` и `false`; поддерживают преобразования типов из `bool` к целому типу и обратно. Ноль эквивалентен `false`, любое другое число эквивалентно `true`.

Ключевое слово `explicit`

Ключевое слово `explicit` запрещает автоматическое преобразование типа для конструктора с одним аргументом. Типичный пример ситуации, в которой это необходимо, — класс коллекции (например, стек), конструктору которого при вызове передается начальный размер коллекции:

```
class Stack {
    explicit Stack(int size); // Создание стека с заданным
    // исходным размером
    ...
};
```

В данном случае ключевое слово `explicit` играет очень важную роль. Без `explicit` этот конструктор автоматически определял бы преобразование типа из `int` в `Stack`. В этом случае объектам типа `Stack` можно было бы присваивать тип `int`:

```
Stack s;
...
s = 40; // Команда создает новый стек на 40 элементов
// и присваивает его s.
```

В результате автоматического преобразования типа число 40 преобразуется в стек из 40 элементов, а результат присваивается `s`. Вряд ли это именно то, что предполагалось. Но если объявить конструктор `int` с ключевым словом `explicit`, попытка присваивания вызовет ошибку на стадии компиляции.

Учтите, что слово `explicit` также исключает инициализацию с автоматическим преобразованием типа при присваивании:

```
Stack s1(40); // OK
Stack s2 = 40; // ОШИБКА
```

Дело в том, что между следующими двумя строками существует тонкое различие:

```
X x;
Y y(x); // Явное преобразование

X x;
Y y = x; // Неявное преобразование
```

В первом фрагменте новый объект типа `Y` создается явным преобразованием от типа `X`, тогда как во втором фрагменте новый объект типа `Y` создается с неявным преобразованием.

Новые операторы преобразования типа

Чтобы программисты могли более четко выразить смысл явного преобразования типа с одним аргументом, в стандарт были включены четыре новых оператора.

О static_cast

Оператор преобразует значение на логическом уровне. Считайте, что он создает временный объект, который затем инициализируется преобразуемыми данными. Преобразование разрешено только при наличии преобразования типа, определенного либо по умолчанию, либо явно. Пример:

```
float x;  
...  
cout << static_cast<int>(x);      // Вывод x в формате int  
...  
f(static_cast<string>("hello"));   // Функция f() вызывается для string.  
                                    // а не для char*
```

О dynamic_cast

Оператор позволяет привести полиморфный тип к «настоящему» статическому типу. Это единственное преобразование типа, которое проверяется на стадии выполнения программы; следовательно, оно может использоваться для проверки типов полиморфных значений. Пример:

```
class Car; // Абстрактный базовый класс  
           // (содержит хотя бы одну чисто виртуальную функцию)  
class Cabriolet : public Car {  
    ...  
};  
  
class Limousine : public Car {  
    ...  
};  
  
void f(Car* cp)  
{  
    Cabriolet* p = dynamic_cast<Cabriolet*>(cp);  
    if (p == NULL) {  
        // p не указывает на объект типа Cabriolet  
        ...  
    }  
}
```

В данном примере в функции `f()` определены специальные действия для тех объектов, которые не относятся к «настоящему» статическому типу `Cabriolet`. Если аргумент является ссылкой, а преобразование типа завершается неудачей, `dynamic_cast` генерирует исключение `bad_cast` (см. с. 43). Помните, что с точки зрения проектирования при работе с полиморфными типами всегда лучше избегать выбора действий в зависимости от фактического типа.

○ `const_cast`

Оператор создает или отменяет атрибут «константности» типа. Кроме того, допускается отмена атрибута `volatile`. Все остальные модификации типа не разрешаются.

○ `reinterpret_cast`

Поведение этого оператора определяется реализацией. Он может (хотя и не обязан) повторно интерпретировать биты, из которых состоит величина. Обычно применение этого преобразования затрудняет переносимость программы.

Перечисленные операторы заменяют старую методику преобразования типов с использованием круглых скобок. К преимуществам нового способа можно отнести то, что он более наглядно выражает намерения программиста при преобразовании. Старый синтаксис применялся во всех рассмотренных случаях, кроме `dynamic_cast`, поэтому, встретив этот оператор в программе, трудно было сразу понять смысл преобразования. Новые операторы также снабжают компилятор дополнительной информацией о причинах преобразования и помогают выявлять ошибки, если преобразование выходит за границы положенного.

Обратите внимание — новые операторы преобразования типа определены только для *одного* аргумента. Рассмотрим следующий пример:

```
static_cast<Fraction>(15,100) // Ошибка: создается Fraction (100)
```

Команда работает совсем не так, как можно было бы ожидать. Вместо того чтобы инициализировать временный объект дроби с числителем 15 и знаменателем 100, она инициализирует временный объект с единственным значением 100. Запятая в данном случае не является разделителем аргументов, а интерпретируется как оператор, который объединяет два выражения и возвращает второе из них. Правильным способом «преобразования» величин 15 и 100 в дробь по-прежнему остается следующая команда:

```
Fraction(15,100) // Правильно, создает Fraction(15,100)
```

Инициализация константных статических членов класса

В новом варианте языка стало возможным инициализировать целочисленные константные статические переменные внутри класса. В частности, это может быть удобно при использовании константы в контексте класса после инициализации. Пример:

```
class MyClass {
    static const int num = 100;
    int elems[num];
    ...
};
```

При этом для константных статических членов, инициализируемых внутри определения класса, все равно приходится резервировать память:

```
const int MyClass::num; // Инициализация не выполняется
```

Определение main

Нужно также прояснить очень важный и часто неверно понимаемый аспект базового языка, а именно правильность функции `main()`. Согласно стандарту C++, переносимыми являются только два определения `main()`:

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

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

Здесь `argv` (массив аргументов командной строки) также может определяться с типом `char**`. Еще обратите внимание на обязательное указание типа возвращаемого значения `int`, поскольку подстановка `int` по умолчанию считается нежелательной.

Функция `main()` может (хотя и не обязана) завершаться командой `return`. В отличие от C в языке C++ `main()` по умолчанию завершается командой

```
return 0
```

Из этого следует, что любая программа, в которой выход из `main()` происходит без команды `return`, завершается успешно (любое значение, отличное от 0, свидетельствует о нестандартном завершении). По этой причине в примерах, приводимых в книге, команда `return` в конце `main()` отсутствует. Учтите, что некоторые компиляторы могут выдавать предупреждение и даже сообщение об ошибке. Что ж, они просто отстали от жизни.

Сложность алгоритмов

В некоторых компонентах стандартной библиотеки C++ (и особенно в STL) первостепенное внимание уделяется скорости работы алгоритмов и функций классов. По этой причине в стандарт включены требования к их «сложности». Специалисты по информатике применяют специальную систему обозначений для сравнения относительной сложности алгоритмов. По этому критерию можно быстро оценить относительное время работы алгоритма, а также сравнить алгоритмы на качественном уровне. Эта система обозначений называется *O-записью*.

О-запись выражает время работы алгоритма как функцию от объема входных данных n . Например, если время работы алгоритма прямо пропорционально количеству элементов (удвоение объема входных данных в два раза увеличивает время работы), сложность алгоритма записывается в виде $O(n)$. Если время работы не зависит от объема входных данных, алгоритм обладает сложностью $O(1)$. В табл. 2.1 перечислены типичные варианты сложности и соответствующая им О-запись.

Таблица 2.1. Типичные варианты сложности

Тип	Обозначение	Описание
Постоянная сложность	$O(1)$	Время работы не зависит от количества элементов
Логарифмическая сложность	$O(\log(n))$	Время работы возрастает в логарифмической зависимости от количества элементов
Линейная сложность	$O(n)$	Время работы возрастает прямо пропорционально количеству элементов
Сложность $n \cdot \log n$	$O(n \cdot \log(n))$	Время работы возрастает как произведение линейной и логарифмической зависимостей от количества элементов
Квадратичная сложность	$O(n^2)$	Время работы возрастает в квадратичной зависимости от количества элементов

Необходимо помнить, что O -запись скрывает менее значимые факторы (например, постоянные затраты времени). Фактическое время работы алгоритма в расчет не принимается; любые два линейных алгоритма считаются эквивалентными. Возможны даже ситуации, когда константа в линейном алгоритме оказывается настолько огромной, что на практике предпочтение может отдаваться экспоненциальному алгоритму с малой константой. Впрочем, O -запись не претендует на совершенство. Просто помните, что она лишь приблизительно характеризует алгоритм, а алгоритм с оптимальной сложностью не всегда является лучшим.

В табл. 2.2 приведены примеры с разными количествами элементов, которые дают представление о темпах увеличения времени работы. Как видите, при малом количестве элементов значения времени работы почти не различаются. Именно здесь постоянные затраты времени, скрытые в O -записи, вносят наибольшее существенный вклад. Но с ростом количества элементов различия начинают проявляться более заметно, а постоянные факторы уходят на второй план. Помните, что при оценке сложности алгоритмов необходимо мыслить масштабно, не ограничиваясь простейшими закономерностями.

Таблица 2.2. Относительное время работы алгоритма в зависимости от типа и количества элементов

Тип	Обозначение	1	2	5	10	50	100	1000
Постоянная сложность	$O(1)$	1	1	1	1	1	1	1
Логарифмическая сложность	$O(\log(n))$	1	2	3	4	6	7	10
Линейная сложность	$O(n)$	1	2	5	10	50	100	1000
Сложность $n \cdot \log n$	$O(n \cdot \log(n))$	1	4	15	40	300	700	10 000
Квадратичная сложность	$O(n^2)$	1	4	25	100	2500	10 000	1000 000

Некоторые определения сложности в справочном руководстве C++ названы *амортизируемыми*. Это означает, что в *долгосрочной перспективе* эти операции

ведут себя так, как описано. С другой стороны, единичные операции могут занимать больше времени. Например, время присоединения элементов к динамическому массиву зависит от того, хватает ли в массиве памяти для очередного элемента. Если памяти достаточно, то операция выполняется с постоянным временем, потому что вставка нового последнего элемента не зависит от размера массива. Но если памяти не хватает, то операция выполняется с линейной сложностью, поскольку для нее приходится выделять новый блок памяти и копировать все элементы. Однако перераспределение памяти происходит относительно редко, поэтому достаточно длинная последовательность операций ведет себя так, как если бы каждая операция выполнялась с постоянной сложностью. Таким образом, вставка характеризуется *амортизируемой постоянной сложностью*.

3 Общие концепции

В этой главе описаны основополагающие концепции стандартной библиотеки C++, необходимые для работы со всеми ее компонентами:

- пространство имен `std`;
- имена и форматы заголовочных файлов;
- общие принципы обработки ошибок и исключений;
- знакомство с распределителями памяти.

Пространство имен `std`

При использовании 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 применяться для разных целей. Для решения этой проблемы в C++ поддерживается концепция *пространств имен* (см. с. 32). Пространство имен определяет область видимости идентификаторов. В отличие от классов пространства имен открыты для расширений, определяемых в любых источниках. Таким образом, в пространстве имен могут определяться компоненты, распределенные по нескольким физическим модулям. Характерным примером такого компонента является стандартная библиотека C++, поэтому вполне логично, что в ней используется пространство имен. Все идентификаторы стандартной библиотеки C++ определяются в пространстве имен с именем `std`.

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

- Непосредственное *уточнение идентификатора* (например, полная запись `std::ostream` вместо `ostream`). В этом варианте записи полная команда вывода может выглядеть так:

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

- Использование *объявления using* (см. с. 33). Например, следующий фрагмент предоставляет локальную возможность пропуска префикса `std::` для `cout` и `endl`:

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

В этом случае предыдущая команда записывается так:

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

- Самый простой вариант — использование директивы `using` (см. с. 33). После выполнения директивы `using` для пространства имен `std` все идентификаторы этого пространства доступны так, как если бы они были объявлены глобально. Например, рассмотрим следующую команду:

```
using namespace std;
```

Эта команда позволит использовать такую запись:

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

Учтите, что в сложных программах это может привести к периодическим конфликтам имен или, что еще хуже, к непредсказуемым последствиям из-за запутанных правил перегрузки. Никогда не используйте директиву `using` при отсутствии полной информации о контексте (например, в заголовочных файлах, модулях или библиотеках).

Примеры, приводимые в книге, невелики, поэтому для простоты в листингах примеров обычно используется последний вариант.

Заголовочные файлы

Все идентификаторы стандартной библиотеки C++ в процессе стандартизации были объединены в пространство имен `std`. Изменения несовместимы 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, проектировщики стремились по возможности сохранить обратную совместимость). Из-за этого пришлось разработать новую схему именования стандартных заголовочных файлов, чтобы для обеспечения совместимости разработчики могли просто включить старые заголовочные файлы.

С определением новых имен для стандартных заголовочных файлов представилась хорошая возможность стандартизировать расширения заголовочных файлов. Раньше на практике использовались разные варианты расширений (например, `.h`, `.hpp` и `.hxx`). Вероятно, спецификации нового стандарта в области заголовочных файлов станут для кого-то неожиданными — теперь стандартные заголовочные файлы вообще не имеют расширений. Команды включения стандартных заголовочных файлов выглядят примерно так:

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

Аналогичное правило действует для заголовочных файлов в стандарте С. Теперь заголовочные файлы С снабжаются префиксом `c` вместо прежнего расширения `.h`:

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

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

Одно из преимуществ новой схемы выбора имен заключается в том, что она позволяет легко отличить старый заголовочный файл для строковых функций `char*` от стандартного заголовочного файла C++ для работы с классом `string`:

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

Впрочем, новая схема именования заголовочных файлов вовсе не означает, что файлы стандартных заголовков не имеют расширений с точки зрения операционной системы. Правила обработки команд `include` для стандартных заголовочных файлов определяются реализацией. Скажем, системы программирования C++ могут добавлять расширения и даже использовать стандартные объявления без загрузки файла. Но на практике большинство систем просто включает заголовок из файла, имя которого точно совпадает с именем, указанным в команде `include`. Следовательно, в большинстве систем стандартные заголовочные файлы C++ не имеют расширений. Помните, что требование об отсутствии расширения относится только к *стандартным* заголовочным файлам. В общем случае рекомендуется снабжать заголовочные файлы расширениями, чтобы упростить их идентификацию в файловой системе.

Для сохранения совместимости с С оставлена поддержка «старых» стандартных заголовочных файлов С. Если потребуется, вы можете использовать команду

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

В этом случае идентификаторы объявляются как в глобальной области видимости, так и в пространстве имен `std`. На практике все выглядит так, словно идентификаторы объявлены в пространстве имен `std`, после чего была выполнена директива `using` (см. с. 33).

Спецификация заголовочных файлов C++ «старого» формата (например, `<iostream.h>`) в стандарте отсутствует; кстати, это решение неоднократно менялось в процессе стандартизации. Таким образом, старые заголовочные файлы официально не поддерживаются. Вероятно, на практике большинство поставщиков будут предоставлять их для обратной совместимости. Однако изменения в заголовках не ограничиваются введением пространства имен `std`, поэтому в общем случае следует либо использовать старые имена заголовочных файлов, либо переключиться на новые стандартизованные имена.

Обработка ошибок и исключений

Стандартная библиотека C++ неоднородна. Она содержит программный код из множества разных источников, отличающихся по стилю проектирования и реализации. Типичным примером таких различий является обработка ошибок и исключений. Одни части библиотеки — например, строковые классы — поддерживают подробную обработку ошибок. Они проверяют все возможные проблемы, которые могут возникнуть в работе программы, и генерируют исключение в случае ошибки. Другие компоненты — например, стандартная библиотека шабло-

нов STL и массивы `valarray` — оптимизируются по скорости, поэтому они редко проверяют логические ошибки и выдают исключения только в случае ошибок времени выполнения.

Стандартные классы исключений

Все исключения, генерируемые языком или библиотекой, происходят от общего предка — базового класса `exception`. Этот класс является корнем иерархического дерева исключений, изображенного на рис. 3.1. Стандартные классы исключений делятся на три категории:

- исключения языковой поддержки;
- исключения стандартной библиотеки C++;
- исключения внешних ошибок.

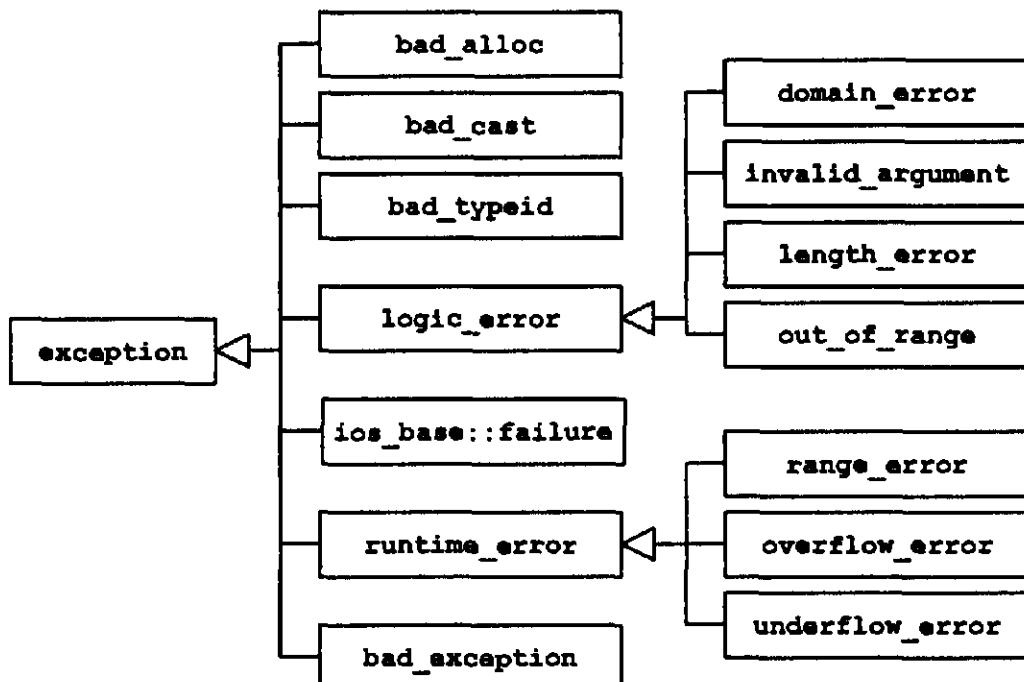


Рис. 3.1. Иерархия стандартных исключений

Классы исключений языковой поддержки

Исключения языковой поддержки используются на уровне языка C++, поэтому их правильнее было бы отнести к базовому языку, нежели к библиотеке. Эти исключения генерируются при неудачных попытках выполнения некоторых операций.

- Исключение класса `bad_alloc` генерируется при неудачном выполнении глобального оператора `new` (кроме версии `new` с запретом исключений). Вероятно, это самое важное исключение, поскольку в любой нетривиальной программе оно может возникнуть в любой момент.
- Исключение класса `bad_cast` генерируется оператором `dynamic_cast`, если преобразование типа по ссылке во время выполнения завершается неудачей. Оператор `dynamic_cast` описан на с. 35.

- Исключение класса `bad_typeid` генерируется оператором `typeid`, предназначенный для идентификации типов во время выполнения. Если аргументом `typeid` является ноль или `null`-указатель, генерируется исключение.
- Исключение класса `bad_exception` предназначено для обработки непредвиденных исключений. В его обработке задействована функция `unexpected()`, которая вызывается при возникновении исключений, не входящих в спецификацию исключений соответствующей функции (спецификации исключений рассматриваются на с. 32). Пример:

```
class E1
class E2

void f() throw(E1)
{
    ...
    throw E1();
    ...
    throw E2();
}
```

Исключение типа `E2` нарушает спецификацию исключений функции `f()`. В этом случае будет вызвана функция `unexpected()`, которая обычно передает управление функции `terminate()` для завершения программы. Но при включении в спецификацию исключений класса `bad_exception` функция `unexpected()` обычно перезапускает исключение этого типа:

```
class E1
class E2

void f() throw(E1, std::bad_exception)
    // Генерирует исключения типа E1 или
    // bad_exception для всех остальных типов исключений
{
    ...
    throw E1(); // Генерирует исключение типа E1
    ...
    throw E2(); // Вызывает функцию unexpected(),
}           // которая генерирует bad_exception
```

Итак, если спецификация исключений содержит класс `bad_exception`, то функция `unexpected()` может заменить исключением `bad_exception`¹ любое исключение, не входящее в спецификацию.

Классы исключений стандартной библиотеки

Классы исключений стандартной библиотеки C++ обычно являются производными от класса `logic_error`. К категории логических ошибок относятся ошибки, которые (по крайней мере, теоретически) можно предотвратить, например, до-

¹ Поведение функции `unexpected()` можно переопределить. Тем не менее функция никогда не генерирует исключения, не указанные в спецификации исключений (если она есть).

полнительной проверкой аргументов функции. В частности, к логическим ошибкам относится нарушение логических предусловий или инварианта класса¹. Стандартная библиотека C++ содержит следующие классы логических ошибок:

- исключение класса `invalid_argument` сообщает о недопустимых значениях аргументов, например, когда битовые поля (массивы битов) инициализируются данными `char`, отличными от 0 и 1;
- исключение класса `length_error` сообщает о попытке выполнения операции, нарушающей ограничения на максимальный размер, например, при присоединении к строке слишком большого количества символов;
- исключение класса `out_of_range` сообщает о том, что аргумент не входит в интервал допустимых значений, например, при использовании неправильного индекса в коллекциях наподобие массивов или в строках;
- исключение класса `domain_error` сообщает об ошибке выхода за пределы области допустимых значений.

Кроме того, в подсистеме ввода-вывода определен специальный класс исключения `ios_base::failure`. Это исключение обычно генерируется при изменении состояния потока вследствие ошибки или достижения конца файла. Поведение этого класса исключений подробно описано на с. 576.

Классы исключений для внешних ошибок

Исключения, производные от `runtime_error`, сообщают о событиях, не контролируемых программой. В стандартную библиотеку C++ включены следующие классы ошибок времени выполнения:

- исключение класса `range_error` сообщает об ошибках выхода за пределы допустимого интервала во внутренних вычислениях;
- исключение класса `overflow_error` сообщает о математическом переполнении;
- исключение класса `underflow_error` сообщает о математической потере значимости.

Исключения, генерируемые стандартной библиотекой

Стандартная библиотека C++ тоже генерирует исключения классов `range_error`, `out_of_range` и `invalid_argument`, а так как библиотека использует языковые средства и пользовательский код, эти исключения могут генерироваться еще и косвенно. В частности, любая операция выделения памяти может привести к исключению `bad_alloc`.

Реализации стандартной библиотеки могут содержать дополнительные классы исключений (определенные на одном уровне со стандартными классами или как производные от них). Однако использование нестандартных классов нарушает переносимость кода, так как замена реализации стандартной библиотеки нарушает работу программы. По этой причине задействовать нестандартные классы исключений нежелательно.

¹ Инвариантом класса называется утверждение, которое должно быть истинным при создании каждого экземпляра объекта класса и сохранять свое значение в течение всего времени жизни объекта. — Примеч. перев.

Заголовочные файлы классов исключений

Базовые классы `exception` и `bad_exception` определяются в заголовочном файле `<exception>`. Класс `bad_alloc` определяется в заголовочном файле `<new>`. Классы `bad_cast` и `bad_typeid` определены в заголовочном файле `<typeinfo>`. Класс `ios_base::failure` определен в заголовочном файле `<iostream>`. Все остальные классы определяются в заголовочном файле `<stdexcept>`.

Члены классов исключений

Обработка исключений в секциях `catch` обычно производится через интерфейс исключений. Интерфейс всех стандартных классов исключений состоит из единственной функции `what()`. Функция возвращает дополнительную информацию (помимо типа исключения) в виде строки, заверпленной нулевым байтом:

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

Содержимое строки определяется реализацией. В частности, это может быть строка с многобайтовой кодировкой, завершенная нулем, интерпретируемая как `wstring` (класс `wstring` описан на с. 463). Стока, возвращаемая функцией `what()`, продолжает существовать вплоть до уничтожения объекта исключения, от которого она была получена¹.

Остальные члены стандартных классов исключений предназначены для создания, копирования, присваивания и уничтожения объектов исключений. Стандартные классы не имеют членов, содержащих дополнительную информацию об исключении (помимо функции `what()`). Например, не существует переносимых средств для получения информации о контексте исключения или неверного индекса при ошибке выхода из интервала. Переносимые возможности обработки дополнительной информации фактически ограничиваются выводом сообщения, полученного от `what()`:

```
try {
    ...
}
catch (const std::exception& error) {
    // Вывод сообщения об ошибке, определяемого реализацией
    std::cerr << error.what() << std::endl;
    ...
}
```

¹ Время жизни возвращаемого значения `what()` в исходном стандарте не оговорено. Для решения проблемы было внесено специальное предложение.

Единственным источником информации об исключении, помимо функции `what()`, является конкретный тип исключения. Например, при обработке исключения `bad_alloc` программа может попытаться выделить дополнительную память.

Генерирование стандартных исключений

Стандартные исключения также могут генерироваться в пользовательских библиотеках или программах. Объекты всех классов стандартных исключений, предоставляющих такую возможность, создаются с одним параметром: объектом `string` (см. главу 11) с описанием, которое должно возвращаться функцией `what()`. Например, определение класса `logic_error` выглядит так:

```
namespace std {  
    class logic_error : public exception {  
        public:  
            explicit logic_error (const string& whatString);  
    };  
}
```

Набор стандартных исключений, которые могут генерироваться в пользовательских библиотеках или программах, включает класс `logic_error` с производными классами, класс `runtime_error` с производными классами, а также класс `ios_base::failure`. Следовательно, вы не сможете генерировать исключения базового класса `exception`, а также любых классов исключений, предоставляющих языковую поддержку.

Чтобы генерировать стандартное исключение, просто создайте строку с описанием и передайте ее объекту исключения в команде `throw`:

```
std::string s;  
...  
throw std::out_of_range(s);
```

Автоматическое преобразование `char*` в `string` позволяет использовать строковый литерал:

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

Классы исключений, производные от `exception`

Существует и другой вариант использования стандартных классов исключений в программах. Он основан на определении специализированных классов, прямо или опосредованно производных от `exception`. При этом необходимо обеспечить работу механизма `what()` в производном классе.

Функция `what()` объявлена виртуальной, поэтому один из вариантов представления функции `what()` основан на ее самостоятельной реализации:

```
namespace MyLib {  
    /* Пользовательский класс исключений  
     * объявлен производным от стандартного класса исключений  
     */
```

```

class MyProblem : public std::exception {
public:
    ...
    MyProblem(...) {           // Специальный конструктор
    }

    virtual const char* what() const throw() { // Функция what()
    ...
    }
};

...
void f() {
    ...
    // Создание объекта и генерирование исключения
    throw MyProblem(...);
    ...
}
}

```

В другом варианте предоставления функции `what()` класс исключения объявляется производным от класса, имеющего конструктор со строковым аргументом для функции `what()`:

```

namespace MyLib {
    /* Пользовательский класс исключения
     * объявлен производным от стандартного класса
     * с конструктором для аргумента what()
     */
    class MyRangeProblem : public std::out_of_range {
public:
    MyRangeProblem (const string& whatString)
        : out_of_range(whatString) {
    }
};

...
void f() {
    ...
    // Создание объекта конструктором со строковым аргументом
    // и генерирование исключения
    throw MyRangeProblem("here is my special range problem");
    ...
}
}

```

Примеры использования этого подхода в программе встречаются в классах `stack` (см. с. 422) и `queue` (см. с. 430).

Распределители памяти

В стандартной библиотеке C++ достаточно часто используются специальные объекты, занимающиеся выделением и освобождением памяти. Такие объекты называются *распределителями* (allocators). Распределитель представляет собой абстракцию, которая преобразует *потребность* в памяти в *физическую операцию* ее выделения. Параллельное использование разных объектов-распределителей позволяет задействовать в программе несколько разных моделей памяти.

Сначала распределители появились в STL для решения раздражающей проблемы с различиями в типах указателей для разных моделей памяти (*near*, *far*, *huge*). В наше время на их основе создаются решения, способные поддерживать разные модели памяти (общая память, уборка «мусора», объектно-ориентированные базы данных) без изменения интерфейса. Впрочем, такое применение распределителей — явление относительно недавнее, и оно еще не получило широкого распространения (вероятно, в будущем ситуация изменится).

В стандартной библиотеке C++ определяется следующий *распределитель по умолчанию*:

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

Этот распределитель используется по умолчанию во всех тех ситуациях, когда распределитель может передаваться в качестве аргумента. Он использует стандартные средства выделения и освобождения памяти, то есть операторы *new* и *delete*, но в спецификации ничего не говорится о том, когда и как эти операторы должны вызываться. Таким образом, конкретная реализация распределителя по умолчанию может, например, организовать внутреннее кэширование выделяемой памяти.

В большинстве программ используется распределитель по умолчанию. Иногда библиотеки предоставляют специализированные распределители, которые просто передаются в виде аргументов. Необходимость в самостоятельном программировании распределителей возникает очень редко, на практике обычно достаточно распределителя по умолчанию. По этой причине мы отложим подробное описание распределителей до главы 15, где рассматриваются не только распределители, но и их интерфейсы.

4 Вспомогательные средства

В этой главе описаны вспомогательные средства стандартной библиотеки C++:

- простые компактные классы и функции для решения распространенных задач;
- типы общего назначения;
- важные функции C;
- числовые пределы¹.

Многие (хотя и не все) из этих средств описаны в секции 20 стандарта C++, посвященной основным утилитам, а их определения находятся в заголовочном файле `<utility>`. Остальные рассматриваются при описании какого-либо из более сложных компонентов библиотеки либо потому, что рассчитаны в основном на работу с этим компонентом, либо по чисто историческим причинам. Например, некоторые вспомогательные функции общего назначения определяются в заголовочном файле `<algorithm>`, хотя они не являются алгоритмами в интерпретации STL (эта тема рассматривается в главе 5).

Некоторые вспомогательные средства также применяются в стандартной библиотеке C++. Например, во всех случаях, когда две величины обрабатываются как единое целое (например, если функция должна вернуть сразу две величины), используется тип `pair`.

Пары

Тип `pair` позволяет работать с двумя величинами как с единственным целым. Он неоднократно встречается в стандартной библиотеке C++. В частности, классы контейнеров `map` и `multimap` используют тип `pair` при выполнении операций с элементами, которые представляют собой пары «ключ/значение» (см. с. 210). Другой распространенный пример – функция, возвращающая два значения.

¹ Вообще говоря, числовые пределы правильнее было бы рассматривать в главе 12, посвященной работе с числовыми данными, однако они используются в других частях библиотеки и поэтому описаны здесь.

Структура `pair` определяется в заголовочном файле `<utility>` следующим образом:

```
namespace std {
    template <class T1, class T2>
    struct pair {
        // Имена типов компонентов
        typedef T1 firstType;
        typedef T2 secondType;

        // Компоненты
        T1 first;
        T2 second;

        /* Конструктор по умолчанию - вызовы T1() и T2()
         * обеспечивают инициализацию базовых типов
         */
        pair()
            : first(T1()), second(T2()) {
        }

        // Конструктор с двумя значениями
        pair(const T1& a, const T2& b)
            : first(a), second(b) {
        }

        // Копирующий конструктор с автоматическими преобразованиями
        template<class U, class V>
        pair(const pair<U, V> &p)
            : first(p.first), second(p.second) {
        }
    };

    // Сравнения
    template <class T1, class T2>
    bool operator== (const pair<T1,T2>&, const pair<T1,T2>&);

    template <class T1, class T2>
    bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);

    ... // И т.д для !=, <=, >, >=

    template <class T1, class T2>
    pair<T1, T2> make_pair(const T1&, const T2&);
}
```

Обратите внимание: тип объявляется с ключевым словом `struct` вместо `class`, а все его члены объявляются открытыми. Следовательно, для любой пары значений возможны прямые обращения к отдельным компонентам.

Конструктор по умолчанию создает пару значений, инициализируемых конструкторами по умолчанию соответствующих типов. Согласно правилам языка, явный вызов конструктора по умолчанию также инициализирует базовые типы

данных (такие, как `int`), поэтому следующее объявление инициализирует компоненты `p` конструкторами `int()` и `float()`, что приводит к обнулению обеих переменных:

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

Инициализация базовых типов вызовом конструктора по умолчанию описана на с. 30.

Шаблонная версия конструктора используется в ситуациях, требующих автоматического преобразования типов. При копировании объекта типа `pair` вызывается стандартный копирующий конструктор¹, генерированный по обычным правилам. Пример:

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

Парные сравнения

Для сравнения двух пар в стандартную библиотеку C++ включены обычные операторы сравнения. Две пары считаются равными при совпадении отдельных компонентов:

```
namespace std {  
    template <class T1, class 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 первый компонент обладает более высоким приоритетом. Если первые компоненты пар различны, то результат их сравнения (больше или меньше) определяет результат сравнения для пар в целом. Если первые компоненты равны, то результат сравнения пар определяется сравнением вторых компонентов:

```
namespace std {  
    template <class T1, class 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 определяются аналогично.

¹ Шаблонный конструктор не замещает стандартный копирующий конструктор. Дополнительная информация по этому вопросу приведена на с. 29.

Вспомогательная функция `make_pair`

Шаблонная функция `make_pair()` позволяет создать пару значений без явного указания типов¹:

```
namespace std {
    // Создание объекта пары только по значениям
    template <class T1, class T2>
    pair<T1, T2> make_pair(const T1& x, const T2& y) {
        return pair<T1,T2>(x, y);
    }
}
```

Например, благодаря функции `make_pair` следующие строки эквивалентны:

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

В частности, функция `make_pair` позволяет легко передать два значения, образующие пару, функции с аргументом типа `pair`. Рассмотрим следующий пример:

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

...
void foo {
    f(std::make_pair(42,"hello")); // Два значения передаются в виде пары
    g(std::make_pair(42,"hello")); // Два значения передаются в виде пары
                                  // с преобразованием типа.
}
```

Как видно из приведенного примера, функция `make_pair()` упрощает передачу двух значений в одном аргументе. Она работает даже при неполном соответствии типов, поскольку шаблонный конструктор обеспечивает автоматическое преобразование. Такая возможность особенно часто используется при работе с отображениями и мультиотображениями (см. с. 210).

Впрочем, у выражений с явным указанием типов есть свое преимущество — они четко определяют тип полученной пары. Например, следующие два выражения дают *разные* результаты:

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

Во втором случае создается объект `pair`, у которого второй компонент относится к типу `double` (вещественные литералы с неуточненным типом интерпретируются как `double`). Тип может оказаться существенным при использовании перегруженных функций или шаблонов, в которых, например, для повышения эффективности могут быть предусмотрены разные версии для `float` и `double`.

¹ Использование функции `make_pair()` не требует дополнительных затрат при выполнении программы. Компилятор всегда оптимизирует подобные вызовы.

Примеры использования пар

Тип `pair` часто встречается в стандартной библиотеке C++. Например, контейнеры `map` и `multimap` используют его для операций с элементами, представляющими пары «ключ/значение». Контейнеры `map` и `multimap` рассматриваются на с. 200, а на с. 103 приведен пример практического использования типа `pair`. Объекты типа `pair` также применяются в функциях стандартной библиотеки C++, возвращающих два значения (см. пример на с. 192).

Класс `auto_ptr`

В этом разделе рассматривается тип `auto_ptr`. Стандартная библиотека C++ предоставляет его как своего рода «умный указатель», помогающий предотвратить утечки ресурсов при исключениях. Обратите внимание на слова «своего рода». Они добавлены, потому что на практике применяется несколько разновидностей умных указателей. Класс `auto_ptr` полезен только при решении определенного круга проблем. За его пределами тип `auto_ptr` не поможет. Будьте внимательны и тщательно изучите материал этого раздела.

Знакомство с классом `auto_ptr`

Функции часто работают по следующей схеме¹.

1. Получение ресурсов.
2. Выполнение операций.
3. Освобождение полученных ресурсов.

Если полученные ресурсы связаны с локальными объектами, они автоматически освобождаются при выходе из функции в результате вызова деструкторов локальных объектов. Но если ресурсы не связаны с объектом, они должны освобождаться явно. Как правило, такие операции с ресурсами выполняются с применением указателей.

Характерный пример работы с ресурсами через указатели — создание и уничтожение объектов операторами `new` и `delete`:

```
void f()
{
    ClassA* ptr = new ClassA;      // Создание объекта
    ...
    delete ptr;                  // Уничтожение объекта
}
```

¹ Объяснение основано на материале книги Скотта Мейерса (Scott Meyers) «More Effective C++». Общая схема была изначально представлена Бьянном Страуструпом под термином «выделение ресурсов при инициализации» в книгах «The C++ Programming Language», 2nd Edition, и «The Design and Evolution of C++». Указатель `auto_ptr` был включен в стандарт специально для поддержки этой методики.

При использовании подобных схем нередко возникают проблемы. Первая и наиболее очевидная проблема заключается в том, что программист может забыть об удалении объекта (особенно если внутри функции присутствуют команды `return`). Но существует и другая, менее очевидная опасность: во время работы функции может произойти исключение, что приведет к немедленному выходу из функции без выполнения оператора `delete`, находящегося в конце тела функции. В результате возникает утечка памяти или, в общем случае, — ресурсов. Для ее предотвращения обычно приходится перехватывать все возможные исключения. Пример:

```
void f()
{
    ClassA* ptr = new ClassA; // Создание объекта

    try {
        ...
        // Работа с объектом
    }
    catch (...) {
        // Для произвольного исключения:
        delete ptr;           // - освободить ресурс
        throw;                // - перезапустить исключение
    }

    delete ptr;             // Нормальное освобождение ресурса
}
```

Освобождение объекта при возникновении исключений приводит к усложнению программы и появлению лишнего кода. Если по этой схеме обрабатывается не один, а два объекта или имеется несколько секций `catch`, программа становится еще запутаннее. В подобных решениях — сложных и чреватых ошибками — проявляется плохой стиль программирования.

В данной ситуации нужен умный указатель, освобождающий данные, на которые он ссылается, при уничтожении самого указателя. Более того, поскольку такой указатель является локальной переменной, он будет уничтожаться автоматически при выходе из функции независимо от причины выхода — нормального завершения или исключения. Класс `auto_ptr` проектировался именно для этих целей.

Указатель `auto_ptr` является *владельцем* объекта, на который он ссылается. В результате уничтожение указателя автоматически приводит к уничтожению объекта. Для работы `auto_ptr` необходимо, чтобы управляемый объект имел только одного владельца.

Ниже приведен предыдущий пример, переработанный с применением `auto_ptr`:

```
// Заголовочный файл для auto_ptr
#include <memory>

void f()
{
    // Создание и инициализация auto_ptr
    std::auto_ptr<ClassA> ptr(new ClassA);

    ...
    // Работа с указателем
}
```

Команда `delete` и секция `catch` стали лишними. Интерфейс указателя `auto_ptr` почти не отличается от интерфейса обычного указателя; оператор `*` производит разыменование объекта, на который ссылается указатель, а оператор `->` предоставляет доступ к членам класса или структуры. Математические операции с указателями (такие, как `++`) для `auto_ptr` не определены. Впрочем, это скорее достоинство, нежели недостаток, потому что вычисления с указателями слишком часто приводят к неприятностям.

Учтите, что тип `auto_ptr` не позволяет инициализировать объект обычным указателем в конструкции присваивания. Инициализация `auto_ptr` должна производиться напрямую по значению¹:

```
std::auto_ptr<ClassA> ptr1(new ClassA);           // OK
std::auto_ptr<ClassA> ptr2 = new ClassA;           // ОШИБКА
```

Передача прав владения в `auto_ptr`

Тип `auto_ptr` поддерживает семантику строгой принадлежности. Иначе говоря, поскольку тип `auto_ptr` удаляет объект, на который он ссылается, этот объект не может «принадлежать» другим объектам. Два и более экземпляра `auto_ptr` не должны одновременно быть владельцами одного объекта. К сожалению, в программе такая ситуация не исключена (например, если два экземпляра `auto_ptr` инициализируются одним и тем же объектом). Программист обязан позаботиться о том, чтобы этого не случилось.

Возникает вопрос: как работают копирующий конструктор и оператор присваивания типа `auto_ptr`? В обычном варианте эти операции копируют данные из одного объекта `auto_ptr` в другой, но в нашем случае это создает ситуацию, при которой один объект принадлежит сразу двум экземплярам `auto_ptr`. Проблема решается просто, но у этого решения есть одно важное следствие: копирующий конструктор и оператор присваивания передают «право владения» тем объектом, на который ссылается `auto_ptr`.

Рассмотрим следующий пример использования копирующего конструктора:

```
// Инициализация auto_ptr новым объектом
std::auto_ptr<ClassA> ptr1(new ClassA);

// Копирование auto_ptr
// - право владения объектом передается от ptr1 к ptr2
std::auto_ptr<ClassA> ptr2(ptr1);
```

После выполнения первой команды объект, созданный оператором `new`, принадлежит `ptr1`. Вторая команда передает право владения от `ptr1` к `ptr2`. Следова-

¹ Между следующими двумя фрагментами существует тонкое различие:

```
X x;
Y y(x);    // Явное преобразование
и
X x;
Y y = x;   // Неявное преобразование
```

тельно, после выполнения второй команды объект, созданный оператором new, принадлежит ptr2, а ptr1 перестает быть владельцем этого объекта. Объект, созданный конструкцией new ClassA, удаляется только один раз — при уничтожении ptr2.

Оператор присваивания поступает аналогичным образом:

```
// Инициализация auto_ptr новым объектом
std::auto_ptr<ClassA> ptr1(new ClassA);
std::auto_ptr<ClassA> ptr2; // Создание другого экземпляра auto_ptr

ptr2 = ptr1; // Присваивание auto_ptr
// - принадлежность объекта передается от ptr1 к ptr2
```

Обратите внимание: смена владельца *не является* простым копированием. Во всех случаях передачи права владения предыдущий владелец (ptr1 в нашем примере) перестает им быть. В результате после передачи права владения предыдущий владелец содержит null-указатель. Подобное поведение серьезно противоречит общим принципам инициализации и присваивания в языках программирования. Копирующий конструктор *модифицирует* объект, используемый для инициализации нового объекта, а оператор присваивания *модифицирует* правую часть операции присваивания. Программист должен сам следить за тем, чтобы программа не пыталась разыменовать экземпляр auto_ptr, переставший владеть объектом и содержащий null-указатель.

Новое значение, присваиваемое auto_ptr, также должно относиться к типу auto_ptr. Присваивание обычных указателей не допускается:

```
std::auto_ptr<ClassA> ptr; // Создание auto_ptr

ptr = new ClassA; // ОШИБКА
ptr = std::auto_ptr<ClassA>(new ClassA); // OK, удаление старого объекта
// и получение нового.
```

Источник и приемник

Из механизма передачи права владения следует возможность особого применения указателей auto_ptr: они могут использоваться функциями для передачи права владения другим функциям. Возможны два разных сценария.

○ Функция является *приемником* данных. Это происходит в том случае, если auto_ptr передается функции по значению. В этом случае параметр вызванной функции становится владельцем auto_ptr. Следовательно, если функция не произведет обратной передачи, объект будет удален при выходе из функции:

```
void sink(std::auto_ptr<ClassA>); // sink() получает право владения
```

○ Функция является *источником* данных. При возврате из auto_ptr право владения возвращенного значения передается вызвавшей функции. Методика продемонстрирована в следующем примере:

```
std::auto_ptr<ClassA> f()
{
```

```

    std::auto_ptr<ClassA> ptr(new ClassA); // ptr становится владельцем
    ...
    return ptr; // Право владения возвращается на сторону вызова
}

void g()
{
    std::auto_ptr<ClassA> p;

    for (int i=0; i<10; ++i) {
        p = f(); // p становится владельцем возвращаемого объекта
        // (предыдущий возвращенный объект уничтожается).
        ...
    } // Удаление последнего объекта, принадлежащего p
}

```

При каждом вызове функция `f()` создает объект оператором `new` и возвращает его (вместе с правом владения) вызывающей стороне. В результате присваивания значения, возвращаемого функцией, право владения передается переменной `p`. При второй, третьей и последующих итерациях присваивание `p` уничтожает объект, ранее принадлежавший `p`. Выход из `g()`, а следовательно уничтожение `p`, приводит к уничтожению последнего объекта, принадлежащего `p`. В любом случае утечка ресурсов невозможна. Даже при возникновении исключения экземпляра `auto_ptr`, владеющий объектом, обеспечит его уничтожение.

Предупреждение

Семантика `auto_ptr` всегда предполагает владение объектом, поэтому не используйте `auto_ptr` в параметрах или возвращаемом значении, если вы не собираетесь передавать право владения. Для примера рассмотрим наивную реализацию функции вывода объекта, на который ссылается `auto_ptr`. Попытка применить такую функцию на практике закончится катастрофой.

```

// ПЛОХОЙ ПРИМЕР
template <class T>
void bad_print(std::auto_ptr<T> p) // p становится владельцем
{ // переданного аргумента
    // Указывает ли p на объект?
    if (p.get() == NULL) {
        std::cout << "NULL";
    }
    else {
        std::cout << *p;
    }
} // Ошибка - объект, на который ссылается p,
   // удаляется при выходе из функции.

```

Каждый раз, когда этой реализации `bad_print` передается аргумент типа `auto_ptr`, принадлежащий ему объект (если он есть) уничтожается. Дело в том, что вместе со значением аргумента параметру `p` передается право владения объектом `auto_ptr`, а объект, принадлежащий `p`, удаляется при выходе из функции. Вряд ли про-

граммист учел такой поворот событий, поэтому, скорее всего, результатом будет фатальная ошибка времени выполнения:

```
std::auto_ptr<int> p(new int);
*p = 42;           // Изменение значения, на которое ссылается p
bad_print(p);     // Удаление данных, на которые ссылается p
*p = 18;          // ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ
```

Может, стоит передавать `auto_ptr` по ссылке? Однако передача `auto_ptr` по ссылке противоречит концепции владения. Нельзя быть полностью уверенным в том, что функция, получающая `auto_ptr` по ссылке, передаст (или не передаст) право владения. Передача `auto_ptr` по ссылке — очень плохое решение, никогда не используйте его в своих программах.

В соответствии с концепцией `auto_ptr` право владения может передаваться функции при помощи константной ссылки. Но такое решение очень опасно, поскольку программисты обычно полагают, что объект, передаваемый по константной ссылке, остается неизменным. К счастью, на поздней стадии проектирования было принято решение, которое сделало применение `auto_ptr` менее рискованным. Благодаря каким-то ухищрениям реализации передача права владения с константными ссылками стала невозможной. Более того, право владения для констант `auto_ptr` вообще не изменяется:

```
const std::auto_ptr<int> p(new int);
*p = 42;           // Изменение значения, на которое ссылается p
bad_print(p);     // ОШИБКА КОМПИЛЯЦИИ
*p = 18;          // OK
```

Такое решение повышает надежность `auto_ptr`. Многие интерфейсы используют константные ссылки для получения данных, которые проходят внутреннее копирование. Собственно говоря, все контейнерные классы (за примерами обращайтесь к главам 6 и 10) стандартной библиотеки C++ ведут себя подобным образом:

```
template <class T>
void container::insert (const T& value)
{
    ...
    x = value;    // Внутреннее копирование
    ...
}
```

Если бы такое присваивание было возможно для `auto_ptr`, то операция присваивания передала бы право владения контейнеру. Но из-за реальной архитектуры `auto_ptr` этот вызов приводит к ошибке на стадии компиляции:

```
контейнер<std::auto_ptr<int> > c;
const std::auto_ptr<int> p(new int);
...
c.insert(p);    // ОШИБКА
...
```

В конечном счете константные объекты `auto_ptr` снижают вероятность не-предвиденной передачи права владения. Каждый раз, когда объект передается через `auto_ptr`, вы можете воспользоваться константным экземпляром `auto_ptr` и тем самым завершить цепочку передачи.

Константность не означает неизменности объекта, принадлежащего `auto_ptr`. Вы не сможете изменить *право владения* для константного экземпляра `auto_ptr`, однако ничто не мешает изменить *значение*, на которое он ссылается. Пример:

```
std::auto_ptr<int> f()
{
    const std::auto_ptr<int> p(new int); // Право владения не передается
    std::auto_ptr<int> q(new int);       // Право владения передается

    *p = 42;                         // OK: изменение значения, на которое ссылается p
    bad_print(p);                   // ОШИБКА КОМПИЛЯЦИИ
    *p = *q;                         // OK: изменение значения, на которое ссылается p
    p = q;                           // ОШИБКА КОМПИЛЯЦИИ
    return p;                         // ОШИБКА КОМПИЛЯЦИИ
}
```

Если `const auto_ptr` передается или возвращается в аргументе, любая попытка присвоить новый объект приводит к ошибке компиляции. В отношении константности `const auto_ptr` ведет себя как константный указатель (`T* const p`), а не как указатель на константу (`const T* p`), хотя синтаксис вроде бы говорит об обратном.

Объекты `auto_ptr` как переменные классов

Использование `auto_ptr` в качестве переменных классов помогает предотвратить утечку ресурсов. Если заменить обычные указатели на `auto_ptr`, отпадет необходимость в деструкторе, поскольку объект будет автоматически удаляться при уничтожении переменной. Кроме того, `auto_ptr` помогает предотвратить утечку ресурсов, обусловленную исключениями во время инициализации объекта. Учите, что деструкторы вызываются только в том случае, если конструирование было завершено. Следовательно, если исключение происходит внутри конструктора, то деструкторы будут вызваны только для полностью сконструированных объектов. Это может привести к утечке ресурсов, например, если первый вызов `new` завершился успешно, а второй вызвал сбой. Пример:

```
class ClassB {
private:
    ClassA* ptr1;           // Переменные-указатели
    ClassA* ptr2;
public:
    // Конструктор, инициализирующий указатели.
    // Если при втором вызове new произойдет исключение,
    // возникнет утечка ресурсов.
    ClassB(ClassA val1, ClassA val2)
        : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {
    }
```

```
// Копирующий конструктор.  
// Исключение при втором вызове new может привести к утечке ресурсов.  
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 предотвращения возможной утечки ресурсов достаточно заменить указатели объектами `auto_ptr`:

```
class ClassB {  
private:  
    const std::auto_ptr<ClassA> ptr1;      // Переменные auto_ptr  
    const std::auto_ptr<ClassA> ptr2;  
public:  
    // Конструктор, инициализирующий auto_ptr.  
    // Утечка ресурсов невозможна.  
    ClassB (ClassA val1, ClassA 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  
    // дает возможность ptr1 и ptr2 уничтожить их объекты).  
    ...  
};
```

Хотя деструктор можно опустить, все равно придется запрограммировать копирующий конструктор и оператор присваивания. По умолчанию они оба пытаются передавать право владения, чего, по всей вероятности, быть не должно. Кроме того, как упоминалось на с. 58, для предотвращения непредвиденной передачи прав владения следует использовать константный объект `auto_ptr`, если объект, на который ссылается `auto_ptr`, не должен изменяться на протяжении его жизненного цикла.

Неправильное использование класса `auto_ptr`

Тип `auto_ptr` создавался для вполне определенных целей, а именно для предотвращения утечки ресурсов при обработке исключений. К сожалению, в прошлом принципы работы `auto_ptr` изменились, а стандартная библиотека не содержит других умных указателей, поэтому программисты часто неправильно используют `auto_ptr`. Ниже приведены рекомендации, которые помогут предотвратить ошибки с применением `auto_ptr`.

- *Указатели `auto_ptr` не могут соавместио владеть объектами.* Экземпляр `auto_ptr` не должен ссылаться на объект, принадлежащий другому экземпляру `auto_ptr` (или другому объекту). В противном случае, если объект будет удален первым указателем, второй указатель начнет ссылаться на уничтоженный объект и любые попытки чтения или записи приведут к катастрофе.
- *Тип `auto_ptr` не поддерживает массивы.* Запрещено использовать тип `auto_ptr` для ссылок на массивы. Дело в том, что объект `auto_ptr` для принадлежащего ему объекта вызывает оператор `delete`, а не `delete[]`. В стандартной библиотеке C++ не существует аналогичного класса, поддерживающего семантику `auto_ptr` для массивов. Вместо этого в библиотеку включены контейнерные классы для работы с наборами данных (см. главу 5).
- *Тип `auto_ptr` не является «универсально умным» указателем.* Тип `auto_ptr` не предназначен для других проблем, при решении которых применяются умные указатели. В частности, `auto_ptr` не подходит для подсчета ссылок (указатели с подсчетом ссылок гарантируют, что объект будет уничтожен только после удаления последнего из нескольких умных указателей, ссылающихся на объект).
- *Тип `auto_ptr` не отвечает требованиям к элементам контейнеров.* Тип `auto_ptr` не удовлетворяет одному из основных требований к элементам стандартных контейнеров, а именно: после копирования или присваивания `auto_ptr` источник и приемник не эквивалентны. Присваивание или копирование объекта `auto_ptr` приводит к модификации экземпляра-источника, потому что операция фактически передает значение, вместо того чтобы его копировать. Следовательно, объекты `auto_ptr` не могут использоваться в качестве элементов стандартных контейнеров. К счастью, архитектура языка и библиотеки предотвращает эту ошибку при компиляции в среде, соответствующей стандарту. К сожалению, неправильное применение `auto_ptr` в отдельных случаях работает, из-за этого неконстантные объекты `auto_ptr` потенциально так же опасны, как обычные указатели. Возможно, вам повезет, и неправильное использование

auto_ptr не приведет к сбою, но на самом деле это скорее невезение, чем везение, — вы даже не узнаете о том, что допустили ошибку.

Реализация умных указателей для подсчета ссылок рассматривается на с. 144 и 226. Такие указатели удобны при совместном использовании элементов в разных контейнерах.

Примеры использования класса auto_ptr

Первый пример демонстрирует передачу права владения при работе с auto_ptr.

```
// util/autoptr1.cpp
#include <iostream>
#include <memory>
using namespace std;

/* Определение оператора вывода для auto_ptr
 * - вывод значения объекта или NULL
 */
template <class T>
ostream& operator<< (ostream& strm, const auto_ptr<T>& p)
{
    // Указывает ли p на объект?
    if (p.get() == NULL) {
        strm << "NULL";           // НЕТ: вывести строку NULL
    }
    else {
        strm << *p;             // ДА: вывести объект
    }
    return strm;
}

int main()
{
    auto_ptr<int> p(new int(42));
    auto_ptr<int> q;

    cout << "after initialization:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;

    q = p;
    cout << "after assigning auto pointers:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;

    *q += 13;                  // Модификация объекта, принадлежащего q
    p = q;
    cout << "after change and reassignment:" << endl;
```

```

cout << " p: " << p << endl;
cout << " q: " << q << endl;
}

```

Результат работы программы выглядит так:

```

after initialization:
p: 42
q: NULL
after assigning auto pointers:
p: NULL
q: 42
after change and reassignment:
p: 55
q: NULL

```

Обратите внимание: второй параметр функции оператора вывода объявлен как константная ссылка, поэтому `auto_ptr` используется без передачи права владения.

Как упоминалось на с. 56, экземпляры `auto_ptr` не могут инициализироваться с присваиванием, а также им не могут присваиваться обычные указатели:

```

std::auto_ptr<int> p(new int(42));           // OK
std::auto_ptr<int> p = new int(42);          // ОШИБКА

p = std::auto_ptr<int>(new int(42));        // OK
p = new int(42);                           // ОШИБКА

```

Дело в том, что конструктор, используемый для создания `auto_ptr` на базе обычного указателя, объявлен с ключевым словом `explicit` (см. с. 34).

Следующий пример демонстрирует поведение константных объектов `auto_ptr`:

```

// util/autoptr2.cpp
#include <iostream>
#include <memory>
using namespace std;

/* Определение оператора вывода для auto_ptr
 * - вывод значения объекта или NULL
 */
template <class T>
ostream& operator<< (ostream& strm, const auto_ptr<T>& p)
{
    // Указывает ли p на объект?
    if (p.get() == NULL) {
        strm << "NULL";           // НЕТ: вывести строку NULL
    }
    else {
        strm << *p;              // ДА: вывести объект
    }
}

```

```
    return strm;
}

int main()
{
    const auto_ptr<int> p(new int(42));
    const auto_ptr<int> q(new int(0));
    const auto_ptr<int> r;

    cout << "after initialization:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;
    cout << " r: " << r << endl;

    *q = *p;
// *r = *p;      // ОШИБКА: неопределенное поведение
    *p = -77;
    cout << "after assigning values:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;
    cout << " r: " << r << endl;

// q = p;        // ОШИБКА компиляции
// r = p;        // ОШИБКА компиляции
}
```

Результат выполнения программы:

```
after initialization:
p: 42
q: 0
r: NULL
after assigning values:
p: -77
q: 42
r: NULL
```

В примере определен оператор вывода для `auto_ptr`, при этом объект `auto_ptr` передается по константной ссылке. Как упоминалось на с. 58, передавать `auto_ptr` функциям не рекомендуется; данная функция является исключением.

Обратите внимание: следующее присваивание является ошибкой:

```
*r = *p;
```

В этом присваивании производится разыменование экземпляра `auto_ptr`, не ссылающегося на объект. В соответствии со стандартом это приводит к непредсказуемым последствиям (например, аварийному завершению программы). Как показывает приведенный пример, вы можете работать с объектами, на которые ссылается константный указатель `auto_ptr`, но не можете изменять принадлежащие им объекты. Даже если бы переменная `r` была объявлена неконстантной, последняя команда была бы невозможной из-за модификации константы `p`.

Подробное описание класса `auto_ptr`

Класс `auto_ptr` объявлен в заголовочном файле `<memory>`:

```
#include <memory>
```

Шаблонный класс `auto_ptr` доступен для всех типов в пространстве имен `std`. Точное объявление `auto_ptr` выглядит так¹:

```
namespace std {
    // Вспомогательный тип, используемый при копировании и присваивании
    template <class Y> struct auto_ptr_ref {};

    template<class T>
    class auto_ptr {
        public:
            // Тип значения
            typedef T element_type;

            // Конструктор
            explicit auto_ptr(T* ptr = 0) throw();

            // Копирующие конструкторы (с неявным преобразованием типа)
            // Обратите внимание - параметр объявлен неконстантным!
            auto_ptr(const auto_ptr&) throw();
            template<class U> auto_ptr(auto_ptr<U>&) throw();

            // Присваивание (с неявным преобразованием типа)
            // Обратите внимание - параметр объявлен неконстантным!

            auto_ptr& operator=(auto_ptr&) throw();
            template<class U>
            auto_ptr& operator=(auto_ptr<U>&) throw();

            // Деструктор
            ~auto_ptr() throw();

            // Обращение по указателю

            T* get() const throw();
            T& operator*() const throw();
            T* operator->() const throw();

            // Освобождение принадлежащего объекта
            T * release ()throw();
            // Повторная инициализация
            void reset(T* ptr = 0) throw();
```

¹ В тексте приводится слегка усовершенствованная версия, в которой исправлены некоторые недочеты стандарта C++ (тип `auto_ptr_ref` объявлен глобальным, а также добавлен оператор присваивания `auto_ptr_ref` в `auto_ptr`; см. с. 69).

```
// Специальные преобразования,
// используемые при копировании и присваивании
public:
    auto_ptr(auto_ptr_ref<T>) throw();
    auto_ptr& operator= (auto_ptr_ref<T> rhs) throw();
    template<class U> operator auto_ptr_ref<U>() throw();
    template<class U> operator auto_ptr <U>() throw();
};

}
```

Ниже приведены подробные описания отдельных членов класса (сокращение *auto_ptr* означает *auto_ptr<T>*). Полный пример реализации класса *auto_ptr* приведен на с. 70.

Определения типов

auto_ptr::element_type

Тип объекта, владельцем которого является *auto_ptr*.

Конструкторы, присваивание и деструкторы

auto_ptr::auto_ptr () throw()

- Конструктор по умолчанию.
- Создает экземпляр *auto_ptr*, не владеющий никаким объектом.
- Инициализирует значение *auto_ptr* нулем.

explicit auto_ptr::auto_ptr (T ptr) throw()*

- Создает экземпляр *auto_ptr*, владеющий объектом, на который ссылается *ptr*, и указывает на него.
- После вызова **this* становится владельцем объекта, на который ссылается *ptr*. Других владельцев быть не должно.
- Если указатель *ptr* не является null-указателем, он должен содержать значение, полученное при вызове *new*, потому что деструктор *auto_ptr* автоматически вызывает *delete* для принадлежащего ему объекта.
- Конструктор не должен вызываться для значения, полученного при создании массива оператором *new []*. С массивами следует использовать контейнерные классы STL, представленные на с. 88.

auto_ptr::auto_ptr (auto_ptr& ap) throw()
template<class U> auto_ptr::auto_ptr (auto_ptr<U>& ap) throw()

- Копирующий конструктор (для неконстантных значений).
- Создает экземпляр *auto_ptr*, к которому переходит право владения объектом, принадлежавшим экземпляру *ap*.
- После выполнения операции экземпляр *ap* перестает быть владельцем объекта, а его значение становится равным null-указателю. Таким образом, в отличие от обычных реализаций копирующего конструктора происходит модификация исходного объекта.

- Обратите внимание: функция перегружается шаблонной функцией класса (см. с. 28). Тем самым обеспечивается возможность автоматического преобразования типа *ap* к типу созданного экземпляра *auto_ptr* (например, указателя *auto_ptr* на объект производного класса в указатель *auto_ptr* на объект базового класса).

- Передача права владения рассматривается на с. 56.

```
auto_ptr& auto_ptr::operator= (auto_ptr& ap) throw()
template<class U> auto_ptr& auto_ptr::operator= (auto_ptr<U>& ap) throw()
```

- Оператор присваивания (для неконстантных значений).
- Удаляет объект, ранее принадлежавший экземпляру, и принимает право владения объектом, ранее принадлежавшим *ap*.
- После выполнения операции экземпляр *ap* перестает быть владельцем объекта, а его значение становится равным null-указателю. Таким образом, в отличие от обычных реализаций оператора присваивания происходит модификация исходного объекта.
- Объект, на который ссылается экземпляр *auto_ptr* в левой части оператора присваивания (**this*), удаляется вызовом *delete*.
- Обратите внимание: функция перегружается шаблонной функцией класса (см. с. 28). Тем самым обеспечивается возможность автоматического преобразования типа *ap* к типу **this* (например, указателя *auto_ptr* на объект производного класса в указатель *auto_ptr* на объект базового класса).
- Передача права владения рассматривается на с. 56.

```
auto_ptr::~auto_ptr () throw()
```

- Деструктор.
- Если экземпляр *auto_ptr* владеет объектом, для этого объекта вызывается *delete*.

Обращение к принадлежащему объекту

```
T* auto_ptr::get () const throw()
```

- Возвращает адрес объекта, владельцем которого является *auto_ptr*.
- Если экземпляр *auto_ptr* не владеет объектом, возвращается null-указатель.
- Вызов *get()* не изменяет прав владения. Иначе говоря, после вызова экземпляр *auto_ptr* продолжает владеть тем же объектом, которым он владел до вызова.

```
T& auto_ptr::operator* () const throw()
```

- Оператор разыменования.
- Возвращает объект, владельцем которого является *auto_ptr*.
- Если экземпляр *auto_ptr* не владеет объектом, вызов приводит к непредсказуемым последствиям (например, аварийному завершению программы).

```
T* auto_ptr::operator-> () const throw()
```

- Оператор используется для обращения к членам объекта, владельцем которого является *auto_ptr*.

- Если экземпляр `auto_ptr` не владеет объектом, вызов приводит к непредсказуемым последствиям (например, аварийному завершению программы).

Модификация принадлежащего объекта

`T* auto_ptr::release () throw()`

- Экземпляр `auto_ptr` перестает быть владельцем объекта.
- Возвращает адрес объекта, принадлежавшего `auto_ptr` перед вызовом (если он был).
- Если перед вызовом экземпляр `auto_ptr` не владел объектом, возвращается `null`-указатель.

`void auto_ptr::reset (T* ptr = 0) throw()`

- Заново инициализирует экземпляр `auto_ptr` по переданному указателю `ptr`.
- Вызывает `delete` для объекта, принадлежавшего экземпляру `auto_ptr` до вызова.
- После вызова объект, на который ссылается `ptr`, принадлежит `*this`. Другого владельца быть не должно.
- Если указатель `ptr` не является `null`-указателем, он должен содержать значение, полученное при вызове `new`, потому что деструктор `auto_ptr` автоматически вызывает `delete` для принадлежащего объекта.
- При вызове `reset()` не должно передаваться значение, полученное при создании массива оператором `new []`. С массивами следует использовать контейнерные классы STL, представленные на с. 88.

Преобразования

Прочие члены класса `auto_ptr` (вспомогательный тип `auto_ptr_ref` и использующие его функции) реализуют довольно хитрые преобразования, позволяющие выполнять операции копирования и присваивания только с неконстантными экземплярами `auto_ptr` (подробности приведены на с. 60). Ниже приводится краткое пояснение¹. Необходимо обеспечить выполнение двух требований.

- Объекты `auto_ptr` должны передаваться функциям и приниматься от них в виде `r`-значений². Поскольку тип `auto_ptr` является классом, это должно делаться с применением конструктора.
- При копировании `auto_ptr` исходный указатель должен лишаться права владения объектом, для чего копия должна модифицировать исходный экземпляр `auto_ptr`.

¹ Спасибо Биллу Гиббонсу (Bill Gibbons) за предоставленный материал.

² Термины «`r`-значение» и «`l`-значение» происходят от конструкции присваивания `выражение1=выражение2`, в которой левосторонний операнд `выражение1` должен представлять собой (модифицируемое) `l`-значение. Тем не менее, `l`-значение правильнее представлять как `значение определителя объекта`, то есть выражение, определяющее объект по имени или адресу (указателю или ссылке). Эти значения не обязаны быть модифицируемыми; например, имя константного объекта является немодифицируемым `l`-значением. Все выражения, не относящиеся к `l`-значениям, являются `r`-значениями. В частности, временные объекты, создаваемые явно (`T()`) или в результате вызова функции, являются `r`-значениями.

Обычный копирующий конструктор может скопировать *r*-значение, но для этого его параметр должен быть объявлен в виде ссылки на коистантный объект. В случае использования обычного конструктора для копирования *auto_ptr* нам придется объявить переменную класса, содержащую фактический указатель, *изменяемой*, чтобы ее можно было модифицировать в копирующем конструкторе. Но тогда вы сможете написать код, который копирует объекты *auto_ptr*, объявленные константными; передача права владения противоречит объявлению этих объектов константными.

Возможен и другой вариант — найти механизм, который бы позволял преобразовать *r*-значение в *l*-значение. Простая операторная функция преобразования к ссылочному типу не подойдет, поскольку такие функции никогда не вызываются для преобразования объекта к его собственному типу (вспомните, что атрибут «ссылочности» не является частью типа). По этой причине был введен класс *auto_ptr_ref*, обеспечивающий механизм преобразования в *l*-значение. Работа этого механизма основана на различиях между перегрузкой (*overloading*) и правилах идентификации аргументов в шаблонах. Эти различия слишком тонки, чтобы использовать их в качестве общего инструмента программирования, но в данном случае они обеспечивают правильную работу класса *auto_ptr*.

Не удивляйтесь, если ваш компилятор еще не различает константные и не-константные объекты *auto_ptr*. Также утите, что в этом случае работа с *auto_ptr* требует еще большей осторожности — малейшая невнимательность легко приводит к непредвиденной передаче прав владения.

Пример реализации класса *auto_ptr*

Ниже приведен пример реализации класса *auto_ptr*, отвечающей требованиям стандарта¹.

```
// util/autoptr.hpp
/* Класс auto_ptr - усовершенствованная реализация,
 * соответствующая стандарту.
 */
namespace std {
    // Вспомогательный тип, используемый при копировании и присваивании
    // (теперь является глобальным)
    template<class Y>
    struct auto_ptr_ref {
        Y* yp;
        auto_ptr_ref (Y* rhs)
            : yp(rhs) {
        }
    };
    template<class T>
    class auto_ptr {

```

¹ Спасибо Грегу Колвину (Greg Colwin) за эту реализацию *auto_ptr*. Утите, что код не полностью соответствует стандарту. Оказалось, что спецификация стандарта содержит ошибки в описании специальных преобразований, выполняемых при использовании *auto_ptr_ref*. В представленной версии эти ошибки исправлены, однако на момент написания книги вопрос о внесении изменений в стандарт еще обсуждался.

```
private:
    T* ap;      // Указывает на принадлежащий объект
public:
    typedef T element_type;

    // Конструктор
    explicit auto_ptr (T* ptr = 0) throw()
        : ap(ptr) {
    }

    // Копирующие конструкторы (с неявным преобразованием типа)
    // Обратите внимание - параметр объявлен неконстантным!
    auto_ptr (auto_ptr& rhs) throw()
        : ap(rhs.release()) {
    }

    template<class Y>
    auto_ptr (auto_ptr<Y>& rhs) throw()
        : ap(rhs.release()) {
    }

    // Операторы присваивания (с неявным преобразованием типа)
    // Обратите внимание - параметр объявлен неконстантным!
    auto_ptr& operator= (auto_ptr& rhs) throw() {
        reset(rhs.release());
        return *this;
    }

    template<class Y>
    auto_ptr& operator= (auto_ptr<Y>& rhs) throw() {
        reset(rhs.release());
        return *this;
    }

    // Деструктор
    ~auto_ptr() throw() {
        delete ap;
    }

    // Обращение по указателю
    T* get() const throw() {
        return ap;
    }

    T& operator*() const throw() {
        return *ap;
    }

    T* operator->() const throw() {
        return ap;
    }

    // Освобождение принадлежащего объекта
```

```

T* release() throw() {
    T* tmp(ap);
    ap = 0;
    return tmp;
}

// Повторная инициализация
void reset (T* ptr=0) throw() {
    if (ap != ptr) {
        delete ap;
        ap = ptr;
    }
}

// Специальные преобразования с применением вспомогательного типа,
// используемые при копировании и присваивании
auto_ptr(auto_ptr_ref<T> rhs) throw()
: ap(rhs.yp) {
}
auto_ptr& operator= (auto_ptr_ref<T> rhs) throw() { // Новый
    reset(rhs.yp);
    return *this;
}
template<class Y>
operator auto_ptr_ref<Y>() throw() {
    return auto_ptr_ref<Y>(release());
}
template<class Y>
operator auto_ptr<Y>() throw() {
    return auto_ptr<Y>(release());
}
}:
}
}

```

Числовые пределы

На практике предельные значения числовых типов зависят от платформы. В стандартной библиотеке C++ они реализованы в шаблоне `numeric_limits`. Числовые пределы заменяют и дополняют обычные препроцессорные константы языка C. Впрочем, эти константы по-прежнему доступны для целочисленных типов в заголовочных файлах `<climits>` и `<limits.h>`, а для вещественных типов — в заголовочный файлах `<float>` и `<float.h>`. У новой концепции числовых пределов есть два достоинства: во-первых, она повышает уровень типовой безопасности, а во-вторых, позволяет программисту писать шаблоны, которые проверяют эти пределы.

Этот раздел посвящен числовым пределам. Однако следует заметить, что всегда рекомендуется писать независимый от платформы код, рассчитанный на

минимальный гарантированный размер данных типа. Минимальные размеры для разных типов перечислены в табл. 4.1.

Таблица 4.1. Минимальный размер базовых типов

Тип	Минимальный размер
char	1 байт (8 бит)
short int	2 байта
int	2 байта
long int	4 байта
float	4 байта
double	8 байт
long double	8 байт

Класс numeric_limits

Обычно шаблоны применяются для реализации некой функциональности в любом типе. Однако шаблоны также могут использоваться для определения общего интерфейса, реализуемого в типах, поддерживающих этот интерфейс. Для этой цели определяются специализированные версии обобщенного шаблона. Ниже показано, как эта методика применяется в шаблоне `numeric_limits`.

- Обобщенный шаблон задает числовые пределы по умолчанию для произвольного типа:

```
namespace std {
    /* Числовые пределы по умолчанию для произвольного типа */
    template <class T>
    class numeric_limits {
        public:
            // Специализация числовых пределов отсутствует
            static const bool is_specialized = false;

            ... // Прочие члены класса, которые не имеют смысла
                // для обобщенных числовых пределов
    };
}
```

Обобщенный шаблон числовых пределов указывает лишь на то, что для типа `T` числовые пределы не определены. Для этого переменной `is_specialized` присваивается значение `false`.

- Специализированные версии шаблона определяют числовые пределы для всех числовых типов:

```
namespace std {
    /* Числовые пределы для int
     */
    template <> class numeric_limits<int> {
```

```

public:
    // Да, для int существует специализация числовых пределов
    static const bool is_specialized = true;

    static T min() throw() {
        return -2147483648;
    }

    static T max() throw() {
        return 2147483647;
    }
    static const int digits = 31;
    ...
};

}

```

На этот раз переменной `is_specialized` присвоено значение `true`, а остальные члены класса используют числовые пределы конкретного типа (`int` в приведенном примере).

Обобщенный шаблон `numeric_limits` и его стандартные специализации содержатся в заголовочном файле `<limits>`. Заголовок включает специализации для всех базовых типов, представляющих числовые значения: `bool`, `char`, `signed char`, `unsigned char`, `wchar_t`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double` и `long double`. Кроме того, аналогичные специализации легко определяются для пользовательских числовых типов.

В табл. 4.2 перечислены все члены класса `numeric_limits<>` с краткими описаниями. В правом столбце приводятся соответствующие константы С, определяемые в заголовочных файлах `<climits>`, `<limits.h>`, `<cfloat>` и `<float.h>`.

Таблица 4.2. Члены класса `numeric_limits<>`

Член класса	Описание	Константы С
<code>is_specialized</code>	Тип поддерживает специализацию числовых пределов	
<code>is_signed</code>	Знаковый тип	
<code>is_Integer</code>	Целочисленный тип	
<code>is_exact</code>	Вычисления производятся без ошибок округления (<code>true</code> для всех целочисленных типов)	
<code>is_bounded</code>	Тип имеет конечный набор допустимых значений (<code>true</code> для всех встроенных типов)	
<code>is_modulo</code>	Поддерживается сложение по модулю (суммирование двух положительных чисел может привести к меньшему результату)	
<code>is_iec559</code>	Тип соответствует стандартам IEC 559 и IEEE 754	

Член класса	Описание	Константы С
min()	Минимальное конечное значение (минимальное нормализованное значение для вещественных типов с денормализацией; имеет смысл, если <code>is_bounded</code> ! <code>is_signed</code>)	INT_MIN, FLT_MIN, CHAR_MIN, ...
max()	Максимальное конечное значение (имеет смысл, если <code>is_bounded</code>)	INT_MAX, FLT_MAX, ...
digits	Символьные, целочисленные типы — количество незнаковых битов; вещественные типы — количество цифр основания <code>radix</code> (см. далее) в мантиссе	CHAR_BIT
digits10	Количество десятичных цифр (имеет смысл, если <code>is_bounded</code>)	FLT_DIG, ...
radix	Целочисленные типы — основание системы счисления, использованной для представления (почти всегда 2); вещественные типы — основание системы счисления, использованием для представления экспоненты	FLT_RADIX
min_exponent	Минимальный отрицательный целый показатель степени с основанием <code>radix</code>	FLT_MIN_EXP, ...
max_exponent	Максимальный положительный целый показатель степени с основанием <code>radix</code>	FLT_MAX_EXP, ...
min_exponent10	Минимальный отрицательный целый показатель степени с основанием 10	FLT_MIN_10_EXP, ...
max_exponent10	Максимальный положительный целый показатель степени с основанием 10	FLT_MAX_10_EXP, ...
epsilon()	Разность между 1 и минимальной величиной, большей 1	FLT_EPSILON, ...
round_style	Стиль округления (см. с. 77)	
round_error()	Оценка максимальной ошибки округления (в соответствии со стандартом ISO/IEC 10967-1)	
has_infinity	Признак наличия представления для положительной бесконечности	
infinity()	Представление положительной бесконечности (если оно есть)	
has_quiet_NaN	Признак наличия пассивного представления NaN («не число»)	
quiet_NaN	Пассивное представление NaN	
has_signaling_NaN	Признак наличия сигнального представления NaN	
signaling_NaN()	Сигнальное представление NaN	
has_denorm	Возможность представления денормализованных значений (с переменным количеством битов экспоненты — см. с. 77)	

Таблица 4.2 (продолжение)

Член класса	Описание	Константы С
has_denorm_loss	Потеря точности обнаруживается как потеря денормализации, а не как неточный результат	
denorm_min()	Минимальное положительное денормализованное значение	
traps	В типе реализован механизм «ловушек»	
tinyness_before	Предельно малые значения обнаруживаются перед округлением	

Ниже приведен один из вариантов полной специализации числовых пределов для зависимого от платформы типа `float`. Кроме того, в нем приведены точные сигнатуры членов.

```
namespace std {
    template<> class numeric_limits<float> {
        public:
            // Да, для float существует специализация числовых пределов
            static const bool is_specialized = true;

            inline static float min() throw() {
                return 1.17549435E-38F;
            }

            inline static float max() throw() {
                return 3.40282347E+38F;
            }

            static const int digits = 24;
            static const int digits10 = 6;

            static const bool is_signed = true;
            static const bool is_integer = false;
            static const bool is_exact = false;
            static const bool is_bounded = true;
            static const bool is_modulo = false;
            static const bool is_iec559 = true;

            static const int radix = 2;

            inline static float epsilon() throw() {
                return 1.19209290E-07F;
            }

            static const float_round_style round_style
                = round_to_nearest;
            inline static float round_error() throw() {
```

```

        return 0.5F;
    }

    static const int min_exponent = -125;
    static const int max_exponent = +128;
    static const int min_exponent10 = -37;
    static const int max_exponent10 = +38;

    static const bool has_infinity = true;
    inline static float infinity() throw() { return ...; }
    static const bool has_quiet_NaN = true;
    inline static float quiet_NaN() throw() { return ...; }
    static const bool has_signaling_NaN = true;
    inline static float signaling_NaN() throw() { return ...; }
    static const float_denorm_style has_denorm = denorm_absent;
    static const bool has_denorm_loss = false;
    inline static float denorm_min() throw() { return min(); }

    static const bool traps = true;
    static const bool tinyness_before = true;
}:
}

```

Обратите внимание: все переменные класса объявлены статическими и константными, чтобы их значения могли определяться на стадии компиляции. В некоторых реализациях значения, возвращаемые функциями, не удается определить на стадии компиляции. Например, при выполнении кода на разных процессорах вещественные числа могут иметь разные значения.

Значения `round_style` перечислены в табл. 4.3, а значения `has_denorm` — в табл. 4.4. К сожалению, переменная `has_denorm` не называется `denorm_style`, что выглядело бы более логично. Это произошло из-за того, что на поздней стадии стандартизации логический тип был заменен перечисляемым. Тем не менее переменная `has_denorm` может использоваться в логическом контексте, потому что в соответствии со стандартом значение `denorm_absent` равно 0 (эквивалент `false`), а `denorm_present` и `denorm_ineterminate` равны соответственно 1 и -1 (оба значения эквивалентны `true`). Следовательно, `has_denorm` может интерпретироваться как логический признак, указывающий, допускает ли данный тип денормализованные значения.

Таблица 4.3. Стили округления в `numeric_limits<>`

Стиль	Описание
<code>round_toward_zero</code>	Округление в направлении нуля
<code>round_to_nearest</code>	Округление до ближайшего представимого значения
<code>round_toward_infinity</code>	Округление в направлении положительной бесконечности
<code>round_toward_neg_infinity</code>	Округление в направлении отрицательной бесконечности
<code>round_ineterminate</code>	Не определено

Таблица 4.4. Стиль денормализации в numeric_limits<>

Стиль денормализации	Описание
denorm_absent	Тип не допускает денормализованные значения
denorm_present	Тип допускает денормализацию до ближайшего представимого значения
denorm_ineterminate	Не определено

Пример использования класса numeric_limits

Следующий пример демонстрирует использование числовых пределов. Он выводит максимальные возможные значения некоторых типов, а также проверяет, является ли тип `char` знаковым.

```
// util/limits1.cpp
#include <iostream>
#include <limits>
#include <string>
using namespace std;

int main()
{
    // Использование текстового представления для bool
    cout << boolalpha;

    // Вывод максимальных значений 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 максимальных значений для вещественных типов
    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;

    // Проверяем, является ли тип char знаковым.
    cout << "is_signed(char): "
        << numeric_limits<char>::is_signed << endl;
    cout << endl;

    // Проверка наличия числовых пределов у типа string
    cout << "is_specialized(string): "
        << numeric_limits<string>::is_specialized << endl;
}
```

Результат выполнения программы зависит от платформы. Один из возможных вариантов выглядит так:

```
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 означает, что для типа `string` числовые пределы не поддерживаются. Впрочем, это вполне логично, поскольку строки не являются числовыми величинами. Однако приведенный пример наглядно показывает, что программист может обратиться с запросом к любому типу и узнать, поддерживает ли он числовые пределы.

Вспомогательные функции

Библиотека алгоритмов (заголовочный файл `<algorithm>`) включает три вспомогательные функции. Первые две функции вычисляют минимум и максимум по двум величинам, а третья меняет местами два значения.

Вычисление минимума и максимума

Функции, вычисляющие минимум и максимум по двум величинам, определяются в заголовочном файле `<algorithm>`:

```
namespace std {
    template <class T>
    inline const T& min (const T& a, const T& b) {
        return b < a ? b : a;
    }

    template <class T>
    inline const T& max (const T& a, const T& b) {
        return a < b ? b : a;
    }
}
```

Если значения равны, функции обычно возвращают первый элемент. Тем не менее работа программы не должна зависеть от данного факта — это считается проявлением плохого стиля программирования.

Обе функции также могут вызываться с дополнительным аргументом, определяющим критерий сравнения:

```
namespace std {
    template <class T, class Compare>
    inline const T& min (const T& a, const T& b, Compare comp) {
        return comp(b,a) ? b : a;
    }

    template <class T, class Compare>
    inline const T& max (const T& a, const T& b, Compare comp) {
        return comp(a,b) ? b : a;
    }
}
```

В качестве дополнительного аргумента передается функция или объект функции, который проверяет, предшествует ли первая величина второй в некотором заданном порядке (объекты функций представлены на с. 134).

Пример использования функции `max()` с передачей функции сравнения:

```
// util/minmax.cpp
#include <algorithm>
using namespace std;

/* Функция сравнивает 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* px = &x;
    int* py = &y;
    int* pmax;

    // Вызов max() с передачей функции сравнения
    pmax = max (px, py, int_ptr_less);
    //...
}
```

Учтите, что определения `min()` и `max()` требуют совпадения типов. Следовательно, эти функции не могут использоваться для объектов разных типов:

```
int i;
long l;
...
l = std::max(i,l);           // ОШИБКА: разнотипные аргументы
```

С другой стороны, вы можете явно задать типы аргументов (а следовательно, и возвращаемого значения):

```
l = std::max<long>(i,l); // OK
```

Перестановка двух значений

Функция `swap()` меняет местами значения двух объектов. Общая реализация `swap()` определяется в заголовочном файле `<algorithm>` следующим образом:

```
namespace std {
    template<class T>
    inline void swap(T& a, T& b) {
        T tmp(a);
        a = b;
        b = tmp;
    }
}
```

Таким образом, чтобы поменять значения двух произвольных переменных `x` и `y`, можно воспользоваться вызовом

```
std::swap(x,y);
```

Разумеется, этот вызов работает только в том случае, если в функции `swap()` возможно конструирование копий и присваивание.

Большим преимуществом `swap()` является то, что программист может определить специальную реализацию для более сложных типов посредством специализации шаблона или перегрузки функции. Специальные реализации экономят время, используя внутреннюю перестановку членов классов вместо присваивания объектов. В частности, такая возможность реализована во всех стандартных контейнерах (см. с. 153) и строках (см. с. 472). Например, реализация `swap()` для простого контейнера, содержащего только массив и количество элементов, выглядит примерно так:

```
class MyContainer {
private:
    int* elems; // Динамический массив элементов
    int numElems; // Количество элементов
public:
    ...
    // Реализация swap()
    void swap(MyContainer& x) {
        std::swap(elems,x.elems);
        std::swap(numElems,x.numElems);
    }
    ...
};

// Перегрузка глобальной версии swap() для данного типа
```

```
inline void swap (MyContainer& c1, MyContainer& c2)
{
    c1.swap(c2); // Вызов специальной реализации swap()
}
```

Таким образом, вызов `swap()` вместо прямой перестановки значений заметно повышает эффективность операции. Всегда определяйте специализированную версию `swap()` для своих типов, если это поможет повысить быстродействие программы.

Вспомогательные операторы сравнения

Четыре шаблонные функции определяют операторы сравнения `!=`, `>`, `<=` и `>=` вызовом операторов `==` и `<`. Определения этих функций в заголовочном файле `<utility>` выглядят так:

```
namespace std {
    namespace rel_ops {
        template <class T>
        inline bool operator != (const T& x, const T& y) {
            return !(x == y);
        }

        template <class T>
        inline bool operator > (const T& x, const T& y) {
            return y < x;
        }

        template <class T>
        inline bool operator <= (const T& x, const T& y) {
            return !(y < x);
        }

        template <class T>
        inline bool operator >= (const T& x, const T& y) {
            return !(x < y);
        }
    }
}
```

Чтобы использовать эти функции, достаточно определить операторы `<` и `==`. При включении пространства имен `std::rel_ops` другие операторы сравнения определяются автоматически. 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; // Получение доступа к !=, > и т. д.
    X x1, x2;
    ...

    if (x1 != x2) {
        ...
    }
    ...

    if (x1 > x2) {
        ...
    }
    ...
}
```

Обратите внимание: операторы определяются в подпространстве имен `std` с именем `rel_ops`. Они выделены в отдельное пространство имен для того, чтобы определение пользовательских операторов в глобальном пространстве имен не приводило к конфликтам даже при предоставлении глобального доступа ко всем идентификаторам пространства имен `std` директивой `using`:

```
using namespace std; // Операторы не переходят
                    // в глобальную область видимости
```

С другой стороны, программисту, желающему получить прямой доступ к операторам, достаточно выполнить следующую директиву и не полагаться на механизм поиска:

```
using namespace std::rel_ops; // Операторы находятся в глобальной
                            // области видимости.
```

В некоторых реализациях шаблоны операторов определяются с двумя разными типами аргументов:

```
namespace std {
    template <class T1, class T2>
    inline bool operator!=(const T1& x, const T2& y) {
        return !(x == y);
    }
    ...
}
```

Преимущество такой реализации — возможность использования разнотипных операндов (при условии совместимости типов). Однако следует учитывать, что подобные реализации не поддерживаются стандартной библиотекой C++, поэтому в случае их применения программа утрачивает переносимость.

Заголовочные файлы `<cstddef>` и `<cstdlib>`

В программах C++ часто используются два заголовочных файла, совместимых с C: `<cstddef>` и `<cstdlib>`. Они представляют собой обновленные версии заголовочных файлов `<stddef.h>` и `<stdlib.h>` языка С и содержат определения некоторых распространенных констант, макросов, типов и функций.

Определения `<cstddef>`

В табл. 4.5 перечислены определения из заголовочного файла `<cstddef>`. Определение `NULL` часто используется для обозначения указателя, ссылающегося на «ничто». Оно соответствует значению 0 (в виде типа `int` или `long`). В языке С значение `NULL` часто определяется как `(void*)0`, но в C++ такое определение неверно — тип `NULL` должен быть целочисленным, иначе присваивание указателю `NULL` станет невозможным. Дело в том, что в C++ не существует автоматического преобразования `void*` в любой другой тип¹. Учтите, что `NULL` также определяется в заголовочных файлах `<cstdio>`, `<cstdlib>`, `<cstring>`, `<ctime>`, `<cwchar>` и `<clocale>`.

Таблица 4.5. Определения `<cstddef>`

Идентификатор	Описание
<code>NULL</code>	«Неопределенное» значение указателя
<code>size_t</code>	Беззнаковый тип для обозначения размеров (например, количества элементов)
<code>ptrdiff_t</code>	Знаковый тип для разности указателей
<code>offsetof()</code>	Смещение члена структуры или объединения

Определения `<cstdlib>`

В табл. 4.6 перечислены важнейшие определения из заголовочного файла `<cstdlib>`. Две константы — `EXIT_SUCCESS` и `EXIT_FAILURE` — определяются для аргумента функции `exit()`, но они также могут использоваться при возврате значения функцией `main()`.

¹ Из-за путаницы с `NULL` некоторые программисты и руководства по стилю программирования рекомендуют вообще отказаться от `NULL` в C++ и использовать 0 или специальную пользовательскую константу (например, `NIL`). Впрочем, автор использует `NULL` в своих программах, в том числе и в примерах этой книги.

Таблица 4.6. Определения <cstdlib>

Идентификатор	Описание
exit (int код)	Выход из программы (с уничтожением статических объектов)
EXIT_SUCCESS	Признак нормального завершения программы
EXIT_FAILURE	Признак ненормального завершения программы
abort()	Аварийное завершение программы (в некоторых системах может привести к сбоям)
atexit (void (*функция)())	Вызов заданной функции при завершении

Функция `atexit()` регистрирует функции, которые должны быть вызваны при завершении программы. Вызов происходит в порядке, обратном порядку регистрации, и без передачи аргументов. Программа может быть завершена как вызовом `exit()`, так и при достижении конца функции `main()()`.

Функции `exit()` и `abort()` позволяют завершить программу в любой функции без возврата в `main()`.

- Функция `exit()` уничтожает все статические объекты, сбрасывает на диск содержимое буферов, закрывает все каналы ввода-вывода и завершает программу с вызовом функций, зарегистрированных функцией `atexit()`. Если в зарегистрированных функциях происходят исключения, вызывается функция `terminate()`.
- Функция `abort()` немедленно прекращает работу программы без выполнения завершающих действий.

Ни одна из этих функций не уничтожает локальные объекты, поскольку раскрутка стека не выполняется. Чтобы обеспечить вызов деструкторов всех локальных объектов, воспользуйтесь исключениями или обычным механизмом возврата и выхода из `main()`.

5 Стандартная библиотека шаблонов

Стандартная библиотека шаблонов (Standard Template Library, STL) занимает центральное место в стандартной библиотеке C++ и оказывает наибольшее влияние на ее общую архитектуру. STL содержит унифицированные средства для работы с коллекциями с применением современных и эффективных алгоритмов. Благодаря STL программисты могут пользоваться новыми разработками в области структур данных и алгоритмов, не разбираясь в принципах их работы.

С точки зрения программиста, STL содержит подборку классов коллекций для различных целей, а также поддерживает ряд алгоритмов для работы с этими коллекциями. Все компоненты STL оформлены в виде шаблонов и поэтому могут использоваться с произвольными типами элементов. Однако библиотека STL делает еще больше: она формирует архитектуру для включения других классов коллекций и алгоритмов, работающих в сочетании с существующими коллекциями и алгоритмами. В конечном итоге STL поднимает C++ на новый уровень абстракции. Забудьте о самостоятельном программировании динамических массивов, связанных списков и двоичных деревьев, а также о программировании разных алгоритмов поиска. Для использования коллекции нужного типа программист просто определяет соответствующий контейнер, а затем лишь вызывает функции и алгоритмы для обработки данных.

Впрочем, за гибкость STL приходится расплачиваться, и прежде всего тем, что библиотека получилось весьма нетривиальной. Из-за этого тема STL рассматривается в нескольких главах книги. В настоящей главе представлены общие принципы устройства STL, а также некоторые приемы программирования, необходимые для работы с библиотекой. В первых примерах показано, как работать с STL и что при этом необходимо учитывать. В главах 6–9 подробно рассматриваются компоненты STL (контейнеры, итераторы, объекты функций, алгоритмы) с дополнительными примерами.

Компоненты STL

Работа STL основана на взаимодействии разных структурных компонентов, среди которых центральное место занимают контейнеры, итераторы и алгоритмы.

○ *Контейнеры* предназначены для управления коллекциями объектов определенного типа. У каждой разновидности контейнеров имеются свои достоин-

ства и недостатки, поэтому существование разных контейнеров отражает различия между требованиями к коллекциям в программах. Контейнеры могут быть реализованы в виде массивов или связанных списков, а каждый элемент может снабжаться специальным ключом.

- *Итераторы* предназначены для перебора элементов в коллекциях объектов (контейнерах или их подмножествах). Главное достоинство итераторов заключается в том, что они предоставляют небольшой, но стандартный интерфейс, подходящий для любого типа контейнера. Например, одна из основных операций этого интерфейса перемещает итератор к следующему элементу коллекции. В программах такая операция выполняется независимо от внутренней структуры коллекции и может применяться как к массиву, так и к дереву. В каждом контейнерном классе определен собственный тип итератора, который делает все необходимое, потому что ему известна внутренняя структура контейнера.
- Интерфейс итераторов имеет много общего с интерфейсом обычных указателей. Увеличение итератора производится оператором `++`, а для обращения к значению, на которое ссылается итератор, используется оператор `*`. Таким образом, итератор можно рассматривать как своего рода умный указатель, который преобразует команду «перейти к следующему элементу» в конкретные действия, необходимые для конкретного типа контейнера.
- *Алгоритмы* предназначены для обработки элементов коллекций. Например, алгоритмы могут выполнять поиск, сортировку, модификацию или просто использовать элементы коллекции для других целей. В работе алгоритмов используются итераторы. Таким образом, алгоритм достаточно запрограммировать только один раз для обобщенного контейнера, потому что интерфейс итераторов является общим для всех контейнеров.

Для повышения гибкости алгоритмам передаются вспомогательные функции, вызываемые в процессе работы алгоритмов. Тем самым общий алгоритм приспосабливается для конкретных целей, даже весьма специфических и сложных. Например, алгоритму можно передать нестандартный критерий поиска или специальную операцию группировки элементов.

Концепция STL основана на разделении данных и операций. Данные находятся под управлением контейнерных классов, а операции определяются адаптируемыми алгоритмами. Итераторы выполняют функции «клей», связывающие эти два компонента. Благодаря им любой алгоритм может работать с любым контейнером (рис. 5.1).

Концепция STL в известном смысле противоречит исходным принципам объектно-ориентированного программирования: STL отделяет друг от друга данные и алгоритмы, вместо того чтобы объединять их. Тем не менее существуют очень веские аргументы в пользу такого решения. Вообще говоря, любая разновидность контейнера может быть объединена с любым алгоритмом, поэтому в результате образуется очень гибкая, но весьма компактная структура.

Среди важнейших особенностей STL следует назвать то, что все компоненты работают с произвольными типами. Как следует из названия (стандартная библиотека шаблонов), все компоненты оформляются в виде шаблонов, подходящих для любого типа (при условии, что этот тип способен выполнять необходимые операции). STL — хороший пример концепции *унифицированного программирования*.

ния. Контейнеры и алгоритмы унифицируются для произвольных типов и классов соответственно.

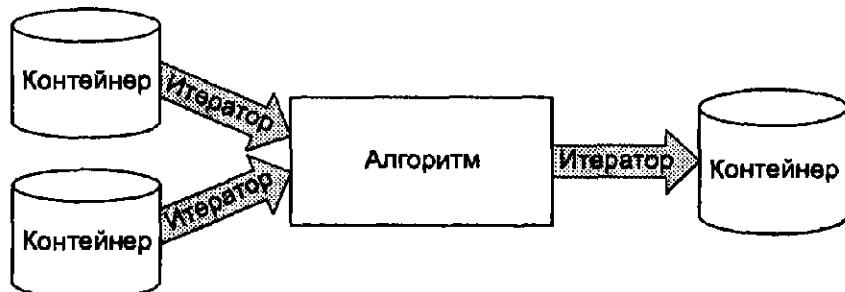


Рис. 5.1. Компоненты STL

Однако в STL присутствуют еще более универсальные компоненты. При помощи *адаптеров* и *объектов функций* (или *функций*) программист может дополнять, ограничивать и настраивать алгоритмы и интерфейсы для конкретных целей. Впрочем, мы опережаем события. Сначала рассмотрим упомянутые концепции на конкретных примерах. Вероятно, это лучший способ понять логику работы STL и познакомиться с библиотекой поближе.

Контейнеры

Контейнерные классы (или проще – *контейнеры*) управляют коллекциями элементов. Для разных потребностей программиста в STL предусмотрены разные типы контейнеров (рис. 5.2).

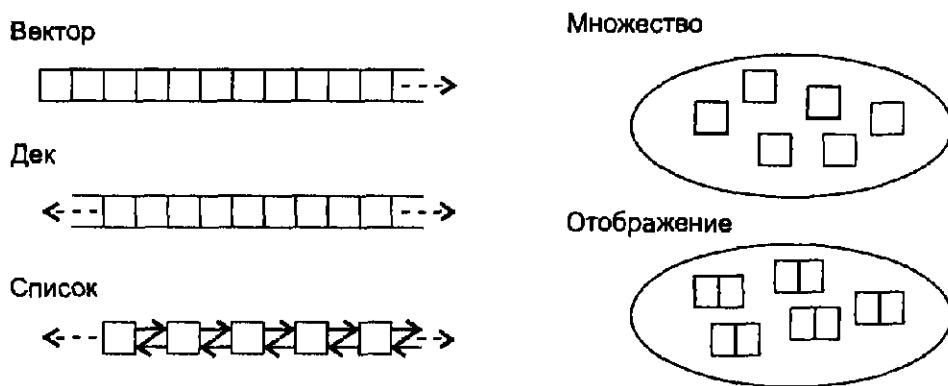


Рис. 5.2. Типы контейнеров STL

Контейнеры делятся на две категории.

- *Последовательные контейнеры* представляют собой *упорядоченные коллекции*, в которых каждый элемент занимает определенную позицию. Позиция зависит от времени и места вставки, но не связана со значением элемента. Например, если последовательно присоединить шесть элементов к концу существующей коллекции, эти элементы будут следовать в порядке их занесения. STL содержит

жит три стандартных класса последовательных контейнеров: `vector` (вектор), `deque` (дек) и `list` (список).

- *Ассоциативные контейнеры* представляют собой *отсортированные коллекции*, в которых позиция элемента зависит от его значения по определенному критерию сортировки. После занесения шести элементов в коллекцию порядок их следования будет определяться только их значениями. Последовательность вставки значения не имеет. STL содержит три стандартных класса ассоциативных контейнеров: `set` (множество), `multiset` (мультимножество), `map` (отображение) и `multimap` (мультиотображение).

Ассоциативный контейнер можно рассматривать как особую разновидность последовательного контейнера, поскольку сортированные коллекции упорядочиваются в соответствии с критерием сортировки. Такой подход вполне естествен для тех, кто работал с другими библиотеками классов коллекций (такими, как в Smalltalk или NIHCL¹), в которых сортированные коллекции были производными от упорядоченных коллекций. Однако не следует забывать, что типы коллекций STL полностью независимы друг от друга. Они имеют разные реализации и не являются производными друг от друга.

Автоматическая сортировка элементов в ассоциативных контейнерах *не означает*, что эти контейнеры специально предназначены для сортировки элементов. С таким же успехом можно отсортировать элементы последовательного контейнера. Основным достоинством автоматической сортировки является более высокая эффективность поиска. В частности, программист всегда может воспользоваться двоичным поиском, для которого характерна логарифмическая, а не линейная сложность. Это означает, что для поиска в коллекции из 1000 элементов в среднем понадобится всего 10 сравнений вместо 500 (см. с. 37). Таким образом, автоматическая сортировка является только (полезным) «побочным эффектом» реализации ассоциативного контейнера, спроектированного с расчетом на повышение эффективности.

В нескольких ближайших разделах подробно рассматриваются разновидности контейнерных классов, и в частности их типичные реализации. Строго говоря, стандартная библиотека C++ не определяет реализацию контейнеров, однако указанные в стандарте поведение и требования к сложности операций не оставляют места для вариаций, поэтому на практике реализации отличаются только во второстепенных деталях. В главе 6 описаны общее поведение контейнерных классов, сходства и различия между ними, а также приводятся подробные описания функций классов.

Последовательные контейнеры

В STL поддерживаются следующие разновидности последовательных контейнеров:

- векторы;
- деки;
- списки.

¹ Библиотека NIHCL (National Institute of Health's Class Library) была одной из первых библиотек классов, написанных на C++.

Кроме того, строки и обычные массивы тоже можно рассматривать как особые разновидности последовательных контейнеров.

Векторы

Вектор управляет элементами, хранящимися в динамическом массиве. Он обеспечивает произвольный доступ к элементам, то есть программа может напрямую обратиться к любому элементу по индексу. Операции присоединения элемента в конец массива и удаления элемента из конца массива выполняются очень быстро¹.

В следующем примере мы определяем вектор для значений типа `int`, вставляем в него шесть элементов и выводим элементы вектора.

```
// stl/vector1.cpp
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> coll; // Вектор с целыми элементами

    // Присоединение элементов со значениями от 1 до 6
    for (int i=1; i<=6; ++i) {
        coll.push_back(i);
    }

    // Вывод элементов, разделенных пробелами
    for (int i=0; i<coll.size(); ++i) {
        cout << coll[i] << ' ';
    }
    cout << endl;
}
```

Следующая директива включает заголовочный файл для работы с векторами:

```
#include <vector>
```

Показанное ниже объявление создает вектор с элементами типа `int`:

```
vector<int> coll;
```

Вектор не инициализируется, поэтому конструктор по умолчанию создает пустую коллекцию.

Функция `push_back()` присоединяет элемент к контейнеру:

```
coll.push_back(i);
```

¹ Строго говоря, присоединение элементов является очень быстрой операцией с учетом амортизации. Отдельные операции могут выполняться медленно, поскольку вектору приходится выделять новую память и копировать в нее существующие элементы. Но так как необходимость в перераспределении памяти возникает довольно редко, в целом операция выполняется очень быстро. Сложность работы алгоритмов рассматривается на с. 37.

Эта функция присутствует во всех последовательных контейнерах.

Функция `size()` возвращает количество элементов в контейнере:

```
for (int i=0; i<coll.size(); ++i) {  
    ...  
}
```

Эта функция поддерживается всеми контейнерными классами.

Оператор индексирования `[]` возвращает один элемент вектора:

```
cout << coll[i] << ' ';
```

В данном примере элементы записываются в стандартный выходной поток данных, поэтому результаты работы программы выглядят так:

```
1 2 3 4 5 6
```

Деки

Термин «дек» (`deque`) происходит от сокращения фразы «double-ended queue» (двусторонняя очередь). Дек представляет собой динамический массив, реализованный таким образом, что может расти в обоих направлениях. Таким образом, операции вставки элементов в конец и в начало коллекции выполняются очень быстро, а вставка в середину занимает больше времени, потому что требует перемещения элементов.

В следующем примере мы объявляем дек для вещественных чисел, вставляем в начало контейнера элементы от 1.1 до 6.6, а затем выводим значения элементов дека.

```
// stl/deque1.cpp  
#include <iostream>  
#include <deque>  
using namespace std;  
  
int main()  
{  
    deque<float> coll; // Дек с вещественными элементами  
  
    // Вставка в начало элементов со значениями от 1.1 до 6.6  
    for (int i=1; i<=6; ++i) {  
        coll.push_front(i*1.1); // Вставка в начало дека  
    }  
  
    // Вывод элементов, разделенных пробелами  
    for (int i=0; i<coll.size(); ++i) {  
        cout << coll[i] << ' ';  
    }  
    cout << endl;  
}
```

Следующая директива включает заголовочный файл для работы с векторами:

```
#include <deque>
```

Показанное ниже объявление создает пустую коллекцию вещественных чисел:

```
deque<float> coll;
```

Функция `push_front()` предназначена для вставки элементов:

```
coll.push_front(i*1.1);
```

Функция `push_front()` вставляет элементы в начало коллекции. Обратите внимание: при таком способе вставки элементы хранятся в контейнере в обратном порядке, потому что каждый новый элемент вставляется перед элементами, вставленными ранее. Следовательно, программа выведет следующий результат:

```
6.6 5.5 4.4 3.3 2.2 1.1
```

Элементы также могут вставляться в конец дека функцией `push_back()`. Функция `push_front()` не поддерживается векторами, поскольку в этом типе контейнера она будет выполняться слишком долго (при вставке элемента в начало вектора придется переместить все существующие элементы). Обычно контейнеры STL содержат только функции, обеспечивающие «хорошую» эффективность (то есть выполняемые с постоянной или логарифмической сложностью). Это сделано для того, чтобы предотвратить случайные вызовы неэффективных функций.

Списки

Класс `list` реализуется в виде двусвязного списка элементов. Это означает, что каждый элемент списка занимает отдельный блок памяти и содержит ссылки на предыдущий и следующий элементы. Списки не поддерживают произвольный доступ к элементам. Например, чтобы обратиться к десятому элементу, необходимо сначала перебрать первые девять элементов по цепочке ссылок. Тем не менее переход к следующему или предыдущему элементу выполняется с постоянным временем, поэтому обобщенная операция доступа к произвольному элементу имеет линейную сложность (среднее расстояние перехода пропорционально количеству элементов). Это гораздо хуже, чем амортизированное постоянное время доступа, обеспечиваемое векторами и деками.

Одним из достоинств списка является быстрота вставки или удаления элементов в любой позиции. Для этого достаточно изменить значения ссылок, поэтому операции вставки и удаления элементов в середине списка выполняются очень быстро по сравнению с аналогичными операциями в векторах или деках.

В следующем примере мы создаем пустой список символов, заносим в него символы от «а» до «z» и выводим все элементы в цикле, который в каждой итерации выводит и удаляет первый элемент коллекции.

```
// stl/list1.cpp
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<char> coll;      // Список с символьными элементами
```

```

// Присоединение элементов от 'а' до 'з'
for (char c='а'; c<='з'; ++c) {
    coll.push_back(c);
}

/* Вывод содержимого списка
 * - пока в списке остаются элементы,
 * - вывести и удалить первый элемент.
 */
while (! coll.empty()) {
    cout << coll.front() << ' ';
    coll.pop_front();
}
cout << endl;
}

```

Как обычно, заголовочный файл `<list>` используется для определения коллекции типа `list`, содержащей символьные элементы:

```
list <char> coll;
```

Функция `empty()` возвращает логический признак отсутствия элементов в контейнере. Цикл выполняется до тех пор, пока функция возвращает `false()`, то есть пока контейнер содержит элементы:

```
while (! coll.empty()) {
    ...
}
```

В теле цикла функция `front()` возвращает первый элемент:

```
cout << coll.front() << ' ';
```

Функция `pop_front()` удаляет первый элемент из контейнера:

```
coll.pop_front();
```

Учтите, что функция `pop_front()` не возвращает удаляемый элемент, поэтому эти две команды не удается заменить одной командой.

Результат выполнения программы зависит от кодировки. В кодировке ASCII он выглядит так¹:

```
a b c d e f g h i j k l m n o p r s t u v w x y z
```

Конечно, процедура «вывода» содержимого цикла, которая выводит и удаляет первый элемент, выглядит несколько странно — обычно в цикле перебираются все элементы. Тем не менее списки не поддерживают произвольный доступ к элементам, поэтому оператор `[]` будет работать недостаточно эффективно. Существует другой способ перебора и вывода элементов, основанный на приме-

¹ В других кодировках результат может содержать символы, не являющиеся буквами, и даже быть пустым (если «z» в этой кодировке не больше «a»).

нении итераторов. Пример будет приведен после знакомства с итераторами (а если вам не терпится, обратитесь к с. 97).

Строки

Строки также могут использоваться в качестве контейнеров STL. Под *строками* подразумеваются объекты строковых классов C++ (`basic_string<>`, `string` и `wstring`), представленные в главе 11. В целом строки аналогичны векторам, но их элементы всегда относятся к символьному типу. Дополнительная информация представлена на с. 480.

Обычные массивы

Другая разновидность контейнеров является не классом, а одним из типов базового языка C и C++: речь идет об обычных массивах со статическим или динамическим размером. Обычные массивы не относятся к контейнерам STL, поскольку они не поддерживают функции типа `size()` или `empty()`, однако архитектура STL позволяет вызывать для них алгоритмы. Это особенно удобно при обработке статических массивов в списках инициализации.

Принципы работы с массивами хорошо известны, нова лишь возможность использования массивов с алгоритмами. Данная тема рассматривается на с. 223.

В C++ необходимость в самостоятельном программировании динамических массивов отпала. Векторы обладают всеми свойствами динамических массивов, но имеют более надежный и удобный интерфейс. Подробности приведены на с. 164.

Ассоциативные контейнеры

Ассоциативные контейнеры автоматически сортируют свои элементы по некоторому критерию. Критерий представляется в виде функции, которая сравнивает либо значения, либо специальные ключи, определяемые для этих значений. По умолчанию элементы или ключи контейнеров сравниваются при помощи оператора `<`. Впрочем, программист может передать в контейнер собственную функцию сравнения и определить новый порядок сортировки.

Ассоциативные контейнеры обычно реализуются в виде бинарных деревьев. У каждого элемента (узла) есть один родитель и два потомка. Все предки слева от узла имеют меньшие значения, а все предки справа — большие значения. Ассоциативные контейнеры различаются по типу элементов и по тому, как они обходятся с дубликатами.

Ниже перечислены стандартные ассоциативные контейнеры, определенные в STL. Для обращения к их элементам применяются итераторы, поэтому примеры откладываются до с. 96, на которой речь пойдет об итераторах.

- *Множества* — коллекции, в которых элементы сортируются в соответствии с их значениями. Каждый элемент присутствует в коллекции только в одном экземпляре, дубликаты не разрешаются.
- *Мульти множества* — то же, что и множества, но с возможностью хранения дубликатов. Это означает, что мульти множество может содержать несколько элементов с одинаковыми значениями.

- *Отображения* — коллекции, состоящие из пар «ключ/значение». У каждого элемента имеется ключ, определяющий порядок сортировки, и значение. Каждый ключ присутствует в коллекции только в одном экземпляре, дубликаты не разрешаются. Отображение также может использоваться как *ассоциативный массив*, то есть массив с произвольным типом индекса (см. с. 103).
- *Мультиотображения* — то же, что и отображения, но с возможностью дублирования ключей. Это означает, что мультиотображение может содержать несколько элементов с одинаковыми ключами. Мультиотображение также может использоваться в качестве *словаря* — см. пример на с. 215.

У всех шаблонов классов ассоциативных контейнеров имеется необязательный аргумент для передачи критерия сортировки. По умолчанию в качестве критерия сортировки используется оператор `<`. Критерий сортировки также применяется при проверке на равенство; два элемента равны, если каждый из них не больше другого.

Множество можно считать особой разновидностью отображения, в котором значение идентично ключу. Все разновидности ассоциативных контейнеров обычно реализуются на базе бинарных деревьев.

Контейнерные адаптеры

Помимо основных контейнерных классов стандартная библиотека C++ содержит специальные контейнерные *адаптеры*, предназначенные для особых целей. В их реализации применяются основные контейнерные классы. Ниже перечислены стандартные контейнерные адаптеры, определенные в библиотеке.

- *Стеки* — контейнеры, элементы которых обрабатываются по принципу LIFO (последним прибыл, первым обслужен).
- *Очереди* — контейнеры, элементы которых обрабатываются по принципу FIFO (первым прибыл, первым обслужен). Иначе говоря, очередь представляет собой обычный буфер.
- *Приоритетные очереди* — контейнеры, элементам которых назначаются приоритеты. Приоритет определяется на основании критерия сортировки, переданного программистом (по умолчанию используется оператор `<`). В сущности, приоритетная очередь представляет собой буфер, следующий элемент которого всегда обладает максимальным приоритетом в очереди. Если максимальный приоритет назначен сразу нескольким элементам, порядок следования элементов не определен.

Исторически контейнерные адаптеры считаются частью STL. Однако с точки зрения программиста, это всего лишь специализированные контейнеры, которые используют общую архитектуру контейнеров, итераторов и алгоритмов, предоставленную STL. По этой причине в книге контейнерные адаптеры рассматриваются отдельно от STL в главе 10.

Итераторы

Итератором называется объект, предназначенный для перебора элементов контейнера STL (всех или некоторого подмножества). Итератор представляет некоторую позицию в контейнере. Ниже перечислены основные операторы, работу с которыми поддерживает итератор.

- `*` — получение элемента в текущей позиции итератора. Если элемент состоит из отдельных членов, для обращения к ним непосредственно через итератор используется оператор `->`.¹
- `++` — перемещение итератора к следующему элементу. Многие итераторы также поддерживают перемещение в обратном направлении, для чего используется оператор `--`.
- `==` и `!=` — проверка совпадений позиций, представленных двумя итераторами.
- `=` — присваивание итератора (позиции элемента, на которую он ссылается).

Этот набор операторов в точности соответствует интерфейсу обычных указателей C и C++ при переборе элементов массива. Различие заключается в том, что итераторы могут быть *умными указателями*, обеспечивающими перебор в более сложных контейнерных структурах данных. Внутреннее поведение итератора зависит от структуры данных, в которой осуществляется перебор. Таким образом, каждая разновидность контейнеров обладает собственным итератором. В каждом контейнерном классе тип итератора определяется в виде вложенного класса. В результате все итераторы обладают одинаковым интерфейсом, но имеют разные типы. В итоге мы приходим к концепции унифицированного программирования: операции выполняются с одинаковым интерфейсом, но с разными типами, что позволяет использовать шаблоны для определения унифицированных операций, работающих с произвольными типами, которые поддерживают заданный интерфейс.

Во всех контейнерных классах поддерживаются базовые функции, применяемые для перебора элементов при помощи итераторов. Ниже перечислены важнейшие из этих функций.

- `begin()` — возвращает итератор, установленный в начало последовательности элементов контейнера. Началом считается позиция первого элемента (если он есть).
- `end()` — возвращает итератор, установленный в конец последовательности элементов контейнера. Концом считается позиция за последним элементом.

Итак, функции `begin()` и `end()` определяют *полуоткрытый интервал*, который содержит первый элемент, но выходит за пределы последнего элемента (рис. 5.3). Полуоткрытый интервал обладает двумя достоинствами:

- появляется простое условие завершения перебора в контейнере: цикл продолжается до тех пор, пока не будет достигнута позиция `end()`;
- предотвращается специальная обработка пустых интервалов, поскольку в пустом интервале `begin()` совпадает с `end()`.

¹ На некоторых старых платформах итераторы еще не поддерживают оператор `->`.

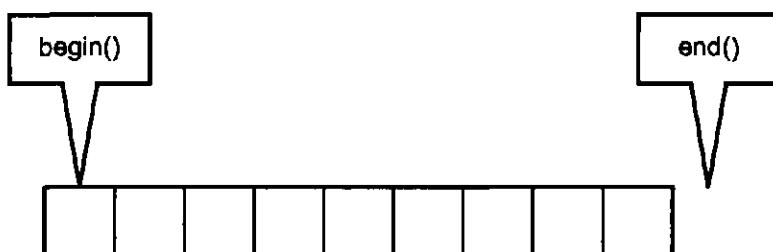


Рис. 5.3. Функции begin() и end()

Следующий пример демонстрирует применение итераторов. В нем выводятся значения всех элементов списка (усовершенствованная версия примера со списками, приведенного на с. 92).

```
// stl/list2.cpp
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<char> coll;      // Список с символьными элементами

    // Присоединение элементов от 'a' до 'z'

    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
    }

    /* Вывод содержимого списка
     * - перебор всех элементов.
     */
    list<char>::const_iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
}
```

После того как список создан и заполнен символами от «a» до «z», все элементы выводятся в цикле for:

```
list<char>::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << *pos << ' ';
}
```

Итератор pos объявляется непосредственно перед началом цикла. В объявлении выбирается тип итератора для обращения к элементам контейнерного класса без возможности модификации:

```
list<char>::const_iterator pos;
```

В каждом контейнере определены два типа итераторов:

- `контейнер::iterator` — используется для перебора элементов в режиме чтения/записи;
- `контейнер::const_iterator` — используется для перебора элементов в режиме чтения.

Скажем, в классе `list` эти определения выглядят примерно так:

```
namespace std {
    template <class T>
    class list {
        public:
            typedef ... iterator;
            typedef ... const_iterator;
            ...
    }
}
```

Конкретный тип итераторов `iterator` и `const_iterator` зависит от реализации.

В теле цикла `for` итератор `pos` инициализируется позицией первого элемента:

```
pos = coll.begin();
```

Цикл продолжается до тех пор, пока `pos` не выйдет за пределы интервала контейнерных элементов:

```
pos != coll.end();
```

В этом условии итератор `pos` сравнивается с конечным итератором. Оператор `++pos` в заголовке цикла перемещает итератор `pos` к следующему элементу.

В итоге итератор `pos` перебирает все элементы, начиная с первого, пока не дойдет до конца (рис. 5.4). Если контейнер не содержит ни одного элемента, тело цикла не выполняется, потому что значение `coll.begin()` будет равным значению `coll.end()`.

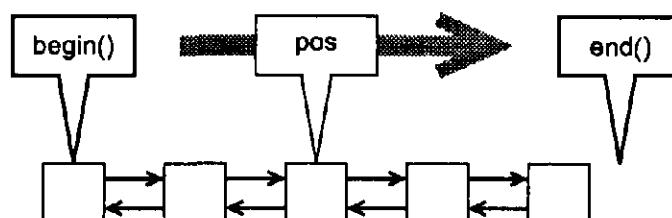


Рис. 5.4. Итератор `pos` при переборе элементов списка

В теле цикла текущий элемент представляется выражением `*pos`. Модификация текущего элемента невозможна, потому что итератор был объявлен с типом `const_iterator`, а значит, с точки зрения итератора элементы являются константными. Но если объявить неконстантный итератор для перебора неконстантных элементов, значения можно будет изменить. Пример:

```
// Приведение всех символов в списке к верхнему регистру
list<char>::iterator pos;
```

```
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    *pos = toupper(*pos);
}
```

Обратите внимание на использование префиксной версии оператора `++`. Возможно, в этом случае она работает эффективнее постфиксной версии, которая создает временный объект для возвращения старой позиции итератора. Из-за этого в общем случае рекомендуется использовать `++pos` вместо `pos++`, а следующее решение нежелательно:

```
for (pos = coll.begin(); pos != coll.end(); pos++) {  
    ...  
    // Работает.  
    // но медленнее  
}
```

По этой причине можно рекомендовать ограничиваться префиксными формами операторов увеличения и уменьшения.

Примеры использования ассоциативных контейнеров

Цикл, приведенный в предыдущем примере, может использоваться с любым типом контейнера — достаточно сменить тип итератора. Ниже приведены примеры использования ассоциативных контейнеров.

Примеры использования множеств и мультимножеств

Первый пример показывает, как происходит вставка элементов в множество и их последующий вывод с использованием итератора:

```
// stl/set1.cpp  
#include <iostream>  
#include <set>  
  
int main()  
{  
    // Тип коллекции  
    typedef std::set<int> IntSet;  
  
    IntSet coll;          // Контейнер для целых чисел  
  
    // Вставка элементов со значениями от 1 до 6  
    * - значение 1 вставляется дважды  
    */  
    coll.insert(3);  
    coll.insert(1);  
    coll.insert(5);  
    coll.insert(4);  
    coll.insert(1);  
    coll.insert(6);
```

```

coll.insert(2);

/* Вывод содержимого множества
 * - перебор всех элементов.
 */
IntSet::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    std::cout << *pos << ' ';
}
std::cout << std::endl;
}

```

Как обычно, следующая директива подгружает все необходимые определения типов и операций с множествами:

```
#include <set>
```

Тип контейнера используется в нескольких местах программы, поэтому для удобства мы определяем для него сокращенное название:

```
typedef std::set<int> IntSet;
```

Команда определяет тип `IntSet` как множество элементов типа `int`. Этот тип использует стандартный критерий сортировки, при котором элементы сортируются оператором `<` (то есть упорядочиваются по возрастанию). Чтобы отсортировать элементы по убыванию или использовать совершенно иной критерий сортировки, передайте его в качестве второго параметра шаблона. Например, следующая команда определяет тип множества с сортировкой элементов по убыванию¹:

```
typedef set<int, greater<int> > IntSet;
```

Стандартный объект функции `greater<>` рассматривается на с. 140, критерий сортировки, использующий только часть данных объекта, — на с. 296.

Все ассоциативные контейнеры поддерживают функцию `insert()`, которая вставляет новый элемент:

```

coll.insert(3);
coll.insert(1);
...

```

Позиция нового элемента определяется автоматически в соответствии с критерием сортировки. Функции последовательных контейнеров `push_back()` или `push_front()` не поддерживаются ассоциативными контейнерами. В данном случае эти функции не имеют смысла, поскольку позиция нового элемента определяется автоматически.

Состояние контейнера после вставки элементов в произвольном порядке иллюстрирует рис. 5.5. В результате сортировки элементы объединяются во

¹ Обратите внимание на пробел между символами `>`. Последовательность `>` воспринимается компилятором как оператор сдвига, что приводит к синтаксической ошибке.

внутреннюю древовидную структуру контейнера таким образом, что «левый» потомок любого элемента всегда меньше (в отношении используемого критерия сортировки) этого элемента, а «правый» потомок всегда больше. Присутствие дубликатов (элементов с одинаковыми значениями) в множествах не допускается, поэтому значение 1 встречается в контейнере только один раз.

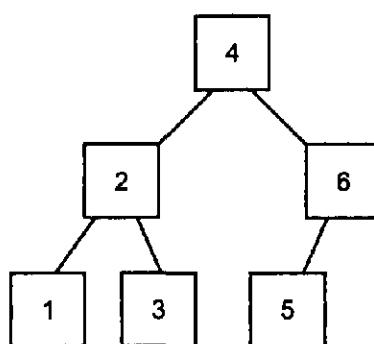


Рис. 5.5. Множество из шести элементов

Вывод элементов контейнера производится в цикле, знакомом по предыдущим примерам. Итератор последовательно перебирает элементы контейнера и выводит их значения:

```

IntSet::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    std::cout << *pos << ' ';
}
  
```

Так как итератор определяется самим контейнером, он справляется со своей задачей, хотя структура контейнера становится более сложной. Например, если итератор ссылается на третий элемент, то оператор `++` переместит его к четвертому (верхнему) элементу. После следующего вызова оператора `++` итератор будет ссылаться на пятый (нижний) элемент (рис. 5.6).

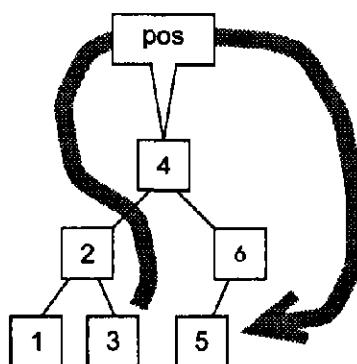


Рис. 5.6. Перебор элементов множества с помощью итератора `pos`

Результат работы программы выглядит так:

1 2 3 4 5 6

Чтобы вместо обычного множества использовалось мультимножество, достаточно изменить тип контейнера (заголовочный файл остается тем же):

```
typedef multiset<int> IntSet;
```

Мультимножество допускает присутствие дубликатов, поэтому контейнер будет содержать два элемента со значением 1. Таким образом, выходные данные программы будут выглядеть так:

```
1 1 2 3 4 5 6
```

Примеры использования отображений и мультиотображений

Элементами отображений и мультиотображений являются пары «ключ/значение», поэтому синтаксис объявления, вставки и обращения к элементам несколько изменяется. Пример использования мультиотображения:

```
// stl/mmap1.cpp
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    // Тип коллекции
    typedef multimap<int,string> IntStringMMap;

    IntStringMMap coll; // Контейнер для хранения пар int/string

    // Вставка элементов в произвольном порядке
    // - значение с ключом 1 вставляется дважды.
    coll.insert(make_pair(5,"tagged"));
    coll.insert(make_pair(2,"a"));
    coll.insert(make_pair(1,"this"));
    coll.insert(make_pair(4,"of"));
    coll.insert(make_pair(6,"strings"));
    coll.insert(make_pair(1,"is"));
    coll.insert(make_pair(3,"multimap"));

    /* Вывод содержимого контейнера
     * - перебор всех элементов
     * - переменная second содержит значение.
     */
    IntStringMMap::iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << pos->second << ' ';
    }
    cout << endl;
}
```

Результат работы программы может выглядеть так:

```
this is a multimap of tagged strings
```

Однако из-за совпадения ключей «this» и «is» эти слова также могут быть выведены в противоположном порядке.

Сравнивая этот пример с приведенным на с. 99, можно заметить два основных различия.

- Элементы представляют собой пары «ключ/значение», поэтому для вставки элемента в коллекцию необходимо создать объект `pair`. Задача решается с помощью вспомогательной функции `make_pair()`. Дополнительная информация вместе с описанием других способов вставки приводится на с. 209.
- Итератор ссылается на пару «ключ/значение», поэтому мы не можем просто направить его в выходной поток данных. Вместо этого приходится работать с отдельными членами структуры `pair`, которым присвоены имена `first` и `second` (тип `pair` представлен на с. 50). Следовательно, выражение `pos->second` определяет второй компонент пары «ключ/значение», то есть значение элемента мультиотображения. Как и в случае с обычными указателями, это выражение представляет собой сокращенную запись для `(*pos).second`.

Аналогичное выражение `pos->first` определяет первый компонент пары «ключ/значение»; в нашем примере это ключ элемента мультиотображения.

Мультиотображения также могут использоваться в качестве *словарей*. Пример приведен на с. 215.

Отображение как ассоциативный массив

Если в предыдущем примере заменить `multimap` на `map`, программа выведет те же данные без дубликатов (значения могут быть теми же). С другой стороны, набор пар «ключ/значение» с уникальными ключами также можно рассматривать как *ассоциативный массив*. Рассмотрим следующий пример:

```
// stl/map1.cpp
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    /* Тип контейнера:
     * - map: элементами являются пары "ключ/значение"
     * - string: ключи относятся к типу string
     * - float: значения относятся к типу float
     */
    typedef map<string,float> StringFloatMap;

    StringFloatMap coll;

    // Вставка элементов в коллекцию
```

```

coll["VAT"] = 0.15;
coll["Pi"] = 3.1415;
coll["an arbitrary number"] = 4983.223;
coll["Null"] = 0;

/* Вывод содержимого коллекции
 * - перебор всех элементов
 * - компонент first содержит ключ
 * - компонент second содержит значение
 */
StringFloatMap::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << "key: \""
        << pos->first << "\" "
        << "value: " << pos->second << endl;
}
}

```

В объявлении типа контейнера должны быть указаны оба типа (ключа и значения):

```
typedef map<string,float> StringFloatMap;
```

При работе с отображениями операция вставки может осуществляться оператором индексирования []:

```

coll["VAT"] = 0.15;
coll["Pi"] = 3.1415;
coll["an arbitrary number"] = 4983.223;
coll["Null"] = 0;

```

Индекс используется в качестве ключа и может иметь произвольный тип. Такой интерфейс типичен для *ассоциативных массивов*. Ассоциативным массивом называется массив, индекс которого может относиться к произвольному типу (не обязательно числовому).

Обратите внимание: оператор индексирования в данном случае работает не так, как обычный оператор индексирования массивов. Отсутствие элемента, связанного с индексом, *не является ошибкой*. При появлении нового индекса (или ключа) создается и вставляется в контейнер новый элемент, ключом которого является указанный индекс. Таким образом, в ассоциативных массивах индекс в принципе не может принимать недопустимое значение. Следовательно, в показанной ниже команде из рассмотренного примера создается новый элемент с ключом "Null":

```
coll["Null"] = 0;
```

Оператор присваивания связывает этот ключ со значением 0, преобразованым к типу float. Использование отображений в качестве ассоциативных массивов более подробно рассматривается на с. 212.

Оператор индексирования не может использоваться с мультиотображениями. Мультиотображение допускает наличие нескольких элементов с одинаковыми

ключами, поэтому оператор индексирования, работающий только с одним значением, в этом случае оказывается бесполезным. Как показывает пример на с. 102, при вставке элементов в мультиотображение необходимо конструировать пары «ключ/значение». Такой способ подходит и для отображений; за подробностями обращайтесь на с. 209.

При обращении к ключу и значению элемента мультиотображения, как и в случае с отображениями, используются компоненты `first` и `second` структуры `pair`. Результат работы программы выглядит так:

```
key: "Null" value: 0
key: "Pi" value: 3.1415
key: "VAT" value: 0.15
key: "an arbitrary number" value: 4983.22
```

Категории итераторов

Возможности итераторов не исчерпываются базовыми операциями. Конкретный состав операций, поддерживаемых итератором, зависит от внутренней структуры типа контейнера. Как обычно, STL поддерживает только те операции, которые выполняются с хорошим быстродействием. Например, если контейнер обеспечивает произвольный доступ, как векторы или деки, его итератор также сможет выполнять операции произвольного доступа, например, его можно установить сразу на пятый элемент.

Итераторы делятся на несколько категорий в зависимости от их общих возможностей. Итераторы стандартных контейнерных классов относятся к одной из двух категорий.

- *Двунаправленные итераторы.* Как подсказывает само название, двунаправленный итератор может перемещаться в двух направлениях: в прямом (оператор `++`) и обратном (оператор `--`). Итераторы контейнерных классов `list`, `set`, `multiset`, `map` и `multimap` являются двунаправленными.
- *Итераторы произвольного доступа.* Итераторы произвольного доступа обладают всеми свойствами двунаправленных итераторов, но в дополнение к ним они обеспечивают произвольный доступ к элементам контейнера. В частности, поддерживаются математические операции с итераторами (по аналогии с математическими операциями над обычными указателями). Вы можете прибавлять и вычитать смещения, обрабатывать разность и сравнивать итераторы с помощью таких операторов, как `<` и `>`. Итераторы контейнерных классов `vector` и `deque`, а также итераторы строк являются итераторами произвольного доступа.

Другие категории итераторов рассматриваются на с. 257.

Чтобы программный код как можно меньше зависел от типа контейнера, постарайтесь по возможности избегать специальных операций с итераторами произвольного доступа. Например, следующий цикл работает с любым контейнером:

```
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    ...
}
```

А этот цикл работает не со всеми контейнерами:

```
for (pos = coll.begin(); pos < coll.end(); ++pos) {  
    ...  
}
```

Единственное различие — наличие оператора `<` вместо `!=` в условии второго цикла. Оператор `<` поддерживается только итераторами произвольного доступа, поэтому второй цикл не будет работать со списками, множествами и отображениями. В унифицированном коде, ориентированном на работу с любыми контейнерами, следует использовать оператор `!=` вместо `<`. С другой стороны, это несколько снижает надежность кода, поскольку `pos` может случайно перейти в позицию за `end()` (некоторые распространенные ошибки, встречающиеся при работе с STL, описаны на с. 146). Сами решайте, какая версия больше подходит для вашего случая. Выбор зависит от контекста, а иногда даже от личных предпочтений.

Чтобы избежать недопонимания, поясняем, что в этом разделе речь идет о *категориях*, а не о *классах* итераторов. Категория определяет только возможности итераторов независимо от их типов. Согласно общей концепции STL, то есть на уровне *чистой абстракции*, любой объект, который *ведет себя* как двунаправленный итератор, является двунаправленным итератором.

Алгоритмы

STL поддерживает несколько стандартных алгоритмов, предназначенных для обработки элементов коллекций. Под обработкой полиморфируется выполнение таких стандартных операций, как поиск, сортировка, копирование, переупорядочение, модификация и численные расчеты.

Алгоритмы не являются функциями контейнерных классов — это глобальные функции, работающие с итераторами. У такого подхода есть одно важное достоинство: вместо того чтобы реализовывать каждый алгоритм для каждого типа контейнера, достаточно реализовать его один раз для обобщенного типа контейнера. Алгоритм может работать даже с элементами контейнеров разных типов, в том числе определенных пользователем. В конечном счете такой подход сокращает объем программного кода, делая библиотеку более мощной и гибкой.

Обратите внимание: речь идет не о парадигме объектно-ориентированного программирования, а об общей парадигме функционального программирования. Вместо того чтобы объединять данные с операциями, как в объектно-ориентированном программировании, мы разделяем их на части, взаимодействующие через некоторый интерфейс. Впрочем, у такого подхода есть свои обратные стороны: во-первых, он недостаточно интуитивен. Во-вторых, некоторые комбинации структур данных и алгоритмов могут оказаться недопустимыми или, что еще хуже, допустимыми, но бесполезными (например, обладающими недостаточным быстродействием). Таким образом, очень важно изучить основные возможности и потенциальные опасности STL, чтобы использовать все достоинства библиотеки и благополучно обойти ловушки. Примеры и дополнительная информация на эту тему неоднократно встречаются в оставшейся части этой главы.

Начнем с простого примера, демонстрирующего некоторые алгоритмы STL и принципы их использования:

```
// stl/algol.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> coll;
    vector<int>::iterator pos;

    // Вставка элементов от 1 до 6 в произвольном порядке
    coll.push_back(2);
    coll.push_back(5);
    coll.push_back(4);
    coll.push_back(1);
    coll.push_back(6);
    coll.push_back(3);

    // Поиск и вывод минимального и максимального элементов
    pos = min_element (coll.begin(), coll.end());
    cout << "min: " << *pos << endl;
    pos = max_element (coll.begin(), coll.end());
    cout << "max: " << *pos << endl;

    // Сортировка всех элементов
    sort (coll.begin(), coll.end());

    // Поиск первого элемента со значением, равным 3
    pos = find (coll.begin(), coll.end(), // Интервал
                3); // Значение

    // Перестановка найденного элемента со значением 3
    // и всех последующих элементов в обратном порядке.
    reverse (pos, coll.end());

    // Вывод всех элементов
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
}
```

Чтобы использовать алгоритмы в программе, необходимо включить в нее заголовочный файл `<algorithm>`:

```
#include <algorithm>
```

Первые два алгоритма называются `min_element()` и `max_element()`. При вызове им передаются два параметра, определяющих интервал обрабатываемых элементов. Чтобы обработать все содержимое контейнера, просто используйте вызовы `begin()` и `end()`. Алгоритмы возвращают итераторы соответственно для минимального и максимального элементов. Таким образом, в показанной ниже команде алгоритм `min_element()` возвращает позицию минимального элемента (если таких элементов несколько, алгоритм возвращает позицию первого из них):

```
pos = min_element (coll.begin(), coll.end());
```

Следующая команда выводит найденный элемент:

```
cout << "min: " << *pos << endl;
```

Разумеется, поиск и вывод можно объединить в одну команду:

```
cout << *min_element(coll.begin(), coll.end(), end()) << endl;
```

Следующим вызывается алгоритм `sort()`. Как нетрудно догадаться по названию, этот алгоритм сортирует элементы в интервале, определяемом двумя аргументами. Как обычно, вы можете задать собственный критерий сортировки. По умолчанию сортировка осуществляется с использованием оператора `<`. Таким образом, следующая команда сортирует все элементы контейнера по возрастанию:

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

В конце концов элементы вектора будут расположены в следующем порядке:

```
1 2 3 4 5 6
```

Алгоритм `find()` ищет значение в заданном интервале. В нашем примере во всем контейнере находится первый элемент со значением 3:

```
pos = find (coll.begin(), coll.end(), // Интервал
            3); // Значение
```

Если поиск оказался успешным, алгоритм `find()` возвращает итератор, установленный на первый найденный элемент. В случае неудачи возвращается итератор для позиции за концом интервала, определяемым вторым переданным аргументом. В нашем примере значение 3 находится в третьем элементе, поэтому `pos` ссылается на третий элемент `coll`.

Последним вызывается алгоритм `reverse()`, который переставляет элементы переданного интервала в обратном порядке. В аргументах передается третий элемент, найденный алгоритмом `find()`, и конечный итератор:

```
reverse (pos, coll.end());
```

В результате вызова переставляются элементы от третьего до последнего. Результат работы программы выглядит так:

```
min: 1
max: 6
1 2 6 5 4 3
```

Интервалы

Любой алгоритм работает с одним или несколькими *интервалами*. Интервал может (хотя и не обязан) содержать все элементы контейнера. Чтобы алгоритм мог обрабатывать подмножество элементов контейнера, при вызове начало и конец интервала передаются в двух разных аргументах (вместо того, чтобы передавать всю коллекцию в одном аргументе).

Такой интерфейс чрезвычайно гибок, но потенциально опасен. Вызывающая сторона должна проследить за тем, чтобы первый и второй аргументы определяли *действительный* интервал, то есть итератор мог перейти от начала к концу интервала в процессе перебора элементов. А это означает, что оба итератора должны принадлежать одному контейнеру, а начало интервала не должно находиться после его конца. Если это условие не выполняется, возможны непредсказуемые последствия, включая зацикливание или нарушение защиты памяти. В этом отношении итераторы так же ненадежны, как обычные указатели. Однако из неопределенности поведения также следует, что реализации STL могут выявлять подобные ошибки и обрабатывать их по своему усмотрению. Как вы вскоре убедитесь, проверить правильность интервала далеко не всегда так просто, как кажется на первый взгляд. За дополнительной информацией о потенциальных проблемах и безопасных версиях STL обращайтесь на с. 146.

Все алгоритмы работают с *полуоткрытыми* интервалами. Иначе говоря, интервал включает заданную начальную позицию, но конечная позиция в него не включается. В традиционной математической имеется два варианта обозначения полуоткрытых интервалов:

```
[начало, конец)  
[начало, конец[
```

В книге используется первый вариант.

Концепция полуоткрытых интервалов обладает рядом достоинств, упоминавшихся на с. 96 (эта концепция проста и не требует специальной обработки пустых коллекций). Тем не менее у нее также есть недостатки. Рассмотрим следующий пример:

```
// stl/find1.cpp  
#include <iostream>  
#include <list>  
#include <algorithm>  
using namespace std;  
  
int main()  
{  
    list<int> coll;  
    list<int>::iterator pos;  
  
    // Вставка элементов от 20 до 40  
    for (int i=20; i<=40; ++i) {  
        coll.push_back(i);  
    }  
}
```

```

// Поиск позиции элемента со значением, равным 3
* - такой элемент отсутствует, поэтому pos присваивается coll.end()
*/
pos = find (coll.begin(), coll.end(), // Интервал
            3);                      // Значение

/* Перестановка элементов от найденного до конца интервала
* - поскольку итератор pos равен coll.end(),
*   перестановка производится в пустом интервале.
*/
reverse (pos, coll.end());

// Поиск позиций со значениями 25 и 25
list<int>::iterator pos25, pos35;
pos25 = find (coll.begin(), coll.end(), // Интервал
              25);                  // Значение
pos35 = find (coll.begin(), coll.end(), // Интервал
              35);                  // Значение

/* Вывод максимума по полученному интервалу
* - обратите внимание: интервал содержит позицию pos25,
*   но позиция pos35 в него не включается.
*/
cout << "max: " << *max_element (pos25, pos35) << endl;

// process the elements including the last position
cout << "max: " << *max_element (pos25, ++pos35) << endl;
}

```

В этом примере коллекция заполняется целыми числами от 20 до 40. После того как поиск элемента со значением 3 завершается неудачей, `find()` возвращает конец обработанного интервала — в данном случае `coll.end()` — и присваивает его переменной `pos`. Использование возвращаемого значения в качестве начала интервала при вызове `reverse()` не вызывает проблем, поскольку это приводит к следующему вызову:

```
reverse (coll.end(), coll.end());
```

Происходит перестановка элементов в пустом интервале, поэтому никакие реальные действия не выполняются.

Но если алгоритм `find()` используется для определения первого и последнего элементов в подмножестве, следует помнить, что при определении интервала на основании полученных итераторов последний элемент исключается. Рассмотрим первый вызов `max_element()`:

```
max_element (pos25, pos35)
```

При этом вызове будет найдено максимальное значение 34 вместо 35:

```
max: 34
```

Чтобы включить в интервал последний найденный элемент, необходимо перевести итератор на одну позицию вперед:

```
max_element (pos25, ++pos35)
```

На этот раз результат будет правильным:

```
max: 35
```

В этом примере контейнером является список, поэтому для перевода `pos35` к следующей позиции был применен оператор `++`. Если бы мы использовали итератор произвольного доступа (как в случае вектора или дека), задача также решалась бы с помощью выражения `pos35 + 1`, поскольку итераторы произвольного доступа поддерживают математические операции (см. с. 105 и 261).

Конечно, итераторы `pos25` и `pos35` можно использовать для поиска в подмножестве элементов. Чтобы поиск производился с включением позиции `pos35`, следует сместить позицию итератора. Пример:

```
// Увеличение pos35 для учета конца интервала при поиске
++pos35;
pos30 = find(pos25, pos35,      // Интервал
             30);            // Значение
if (pos30 == pos35) {
    cout << "30 is NOT in the subrange" << endl;
}
else {
    cout << "30 is in the subrange" << endl;
}
```

Все примеры, приведенные в этом разделе, работают только потому, что нам точно известно: позиция `pos25` предшествует позиции `pos35`. В противном случае интервал `[pos25, pos35)` становится недействительным. Если вы не уверены в том, какой из элементов которому предшествует, то ситуация усложняется, а ошибка может привести к непредсказуемым последствиям.

Предположим, мы не знаем, что элемент со значением 25 предшествует элементу со значением 35. Более того, нельзя исключать, что одно или оба значения отсутствуют в контейнере. При использовании итератора произвольного доступа проверка выполняется оператором `<<`:

```
if (pos25 < pos35) {
    // Действителен интервал [pos25, pos35)
    ...
}
else if (pos35 < pos25) {
    // Действителен интервал [pos35, pos25)
    ...
}
else {
    // Итераторы равны; следовательно, оба итератора
    // находятся в позиции end()
    ...
}
```

Без итераторов произвольного доступа не существует простого и быстрого способа определить порядок следования итераторов. Единственное разумное решение — провести поиск одного итератора в интервале от начала до другого итератора или от другого итератора до конца. В этом случае алгоритм слегка изменяется: вместо того чтобы искать оба значения во всем исходном интервале, мы пытаемся определить, какое значение встречается первым. Пример:

```

pos25 = find (coll.begin(), coll.end(), // Интервал
              25);                      // Значение
pos35 = find (coll.begin(), pos25,      // Интервал
              35);                      // Значение
if (pos35 != pos25) {
    /* pos35 предшествует pos25; следовательно.
     * действителен только интервал [pos35, pos25)
     */
    ...
}
else {
    pos35 = find (pos25, coll.end(), // Интервал
                  35);                      // Значение
    if (pos35 != pos25) {
        /* pos25 предшествует pos35; следовательно.
         * действителен только интервал [pos25, pos35)
         */
        ...
    }
    else {
        // Итераторы равны; следовательно, оба итератора
        // находятся в позиции end()
        ...
    }
}

```

В отличие от предыдущей версии поиск значения 35 производится не во всем множестве элементов `coll`. Сначала поиск ведется от начала до `pos25`. Если значение не найдено, поиск производится среди элементов, находящихся после `pos25`. В результате мы выясняем, какой итератор встречается раньше и какой интервал действителен.

Эффективность подобной реализации оставляет желать лучшего. Более эффективный способ найти первый элемент со значением 25 или 35 заключается в том, чтобы поиск выполнялся именно по такому условию. Для этого придется задействовать некоторые возможности STL, еще не упоминавшиеся в книге:

```

pos = find_if (coll.begin(), coll.end(),           // Интервал
               compose_f_gx_hx(logical_or<bool>(),
                                bind2nd(equal_to<int>(), 25),
                                bind2nd(equal_to<int>(), 35)));
switch (*pos) {
    case 25:
        // Первым обнаружен элемент со значением 25

```

```
pos25 = pos;
pos35 = find (++pos, coll.end(),      // Интервал
              35);                  // Значение
...
break;
case 35:
// Первым обнаружен элемент со значением 35
pos35 = pos;
pos25 = find (++pos, coll.end(),      // Интервал
              25);                  // Значение
...
break;
default:
// Элементы со значениями 25 и 35 не найдены
...
break;
}
```

В качестве критерия поиска передается специальное выражение для нахождения первого элемента со значением 25 или 35. В этом выражении объединены несколько стандартных объектов функций (см. с. 140 и 305) и вспомогательный объект функции `compose_f_gx_hx`, описанный на с. 316.

Использование нескольких интервалов

Некоторые алгоритмы работают сразу с несколькими интервалами. Обычно в таких случаях задаются начало и конец только одного интервала, а для остальных интервалов задается только начало. Конечная позиция других интервалов определяется по количеству элементов в первом интервале. Например, следующий вызов `equal()` поэлементно сравнивает все содержимое коллекции `coll1` с элементами `coll2`, начиная с первого:

```
if (equal (coll1.begin(), coll1.end(),
           coll2.begin())) {
    ...
}
```

Таким образом, количество элементов `coll2`, сравниваемых с элементами `coll1`, косвенно определяется количеством элементов в `coll1`.

Из этого следует важный вывод: вызывая алгоритм для нескольких интервалов, убедитесь в том, что второй и все прочие интервалы содержат не меньше элементов, чем первый интервал. Размер приемных интервалов особенно важен для алгоритмов, осуществляющих запись в коллекцию.

Рассмотрим следующую программу:

```
// stl/copy1.cpp
#include <iostream>
#include <vector>
#include <list>
```

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

int main()
{
    list<int> coll1;
    vector<int> coll2;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll1.push_back(i);
    }

    // ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ:
    // - перезапись несуществующих элементов в приемнике
    copy (coll1.begin(), coll1.end(),           // Источник
          coll2.begin());                      // Приемник
    //...
}
```

В приведенном примере вызывается алгоритм `copy()`, который просто копирует все элементы первого интервала во второй (приемный) интервал. Как обычно, для первого интервала определяются обе границы, а для второго – только начало. Но данный алгоритм производит перезапись элементов вместо вставки, поэтому он *требует*, чтобы количество элементов в присущем интервале было достаточным для перезаписи. Если элементов не хватит (как в приведенном примере), последствия будут непредсказуемыми. На практике это часто означает перезапись данных, находящихся в памяти после `coll2.end()`. Если повезет, произойдет сбой программы; по крайней мере, вы будете знать, что случилось что-то не то. Впрочем, вы можете упростить себе жизнь и воспользоваться безопасной версией STL, в которой неопределенное поведение автоматически приводит к выполнению некоторой процедуры обработки ошибок (см. с. 146).

Чтобы избежать подобных ошибок, либо убедитесь, что приемный интервал имеет достаточные размеры, либо воспользуйтесь *операторами вставки*. Итераторы вставки рассматриваются на с. 116. Сначала посмотрим, как изменить приемный интервал и обеспечить наличие в нем необходимого количества элементов.

Чтобы приемный интервал был достаточно большим, необходимо либо создать его с нужным размером, либо изменить размер в программе. Оба варианта подходят только для последовательных контейнеров (векторов, деков и списков). Впрочем, это не создает особых проблем, поскольку ассоциативные контейнеры не могут использоваться в качестве приемника для алгоритмов с перезаписью (причины объясняются на с. 125). Пример увеличения размера контейнера приводится ниже.

```
// stl/copy2.cpp
#include <iostream>
#include <vector>
#include <list>
```

```
#include <deque>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll1;
    vector<int> coll2;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll1.push_back(i);
    }

    // Изменение размера приемного интервала,
    // чтобы он был достаточен для работы алгоритма
    // с перезаписью.
    coll2.resize (coll1.size());

    /* Копирование элементов из первой коллекции во вторую
     * - перезапись существующих элементов в приемном интервале
     */
    copy (coll1.begin(), coll1.end(),           // Источник
          coll2.begin());                      // Приемник

    /* Создание третьей коллекции с необходимым количеством элементов
     * - исходный размер передается в параметре
     */
    deque<int> coll3(coll1.size());

    // Копирование элементов из первой коллекции в третью
    copy (coll1.begin(), coll1.end(),           // Источник
          coll3.begin());                      // Приемник
}
```

Размеры существующего контейнера `coll2` изменяются функцией `resize()`:

```
coll2.resize (coll1.size());
```

Коллекция `coll3` инициализируется с явным заданием начального размера, чтобы количество элементов в ней соответствовало количеству элементов в `coll1`:

```
deque<int> coll3(coll1.size());
```

Как при изменении размеров, так и при инициализации с заданным размером создаются новые элементы. Эти элементы инициализируются конструктором по умолчанию, поскольку их создание происходит без передачи аргументов. Передача дополнительного аргумента конструктору и функции `resize()` позволяет инициализировать новые элементы.

Итераторные адаптеры

Итераторы являются *чистыми абстракциями*; иначе говоря, любой объект, который *ведет себя* как итератор, является итератором. По этой причине вы можете написать класс, который обладает интерфейсом итератора, но делает нечто совершенно иное. Стандартная библиотека C++ содержит несколько готовых специализированных итераторов, называемых *итераторными адаптерами*. Не стоит полагать, что итераторные адаптеры являются обычными вспомогательными классами; они наделяют саму концепцию итераторов рядом новых возможностей.

В следующих подразделах представлены три разновидности итераторных адаптеров:

- итераторы вставки;
- потоковые итераторы;
- обратные итераторы.

Дополнительная информация приводится на с. 270.

Итераторы вставки

Первая разновидность итераторных адаптеров — *итераторы вставки* — позволяет использовать алгоритмы в режиме вставки (вместо режима перезаписи). В частности, итераторы вставки решают проблему с нехваткой места в приемном интервале при записи: приемный интервал просто увеличивается до нужных размеров.

Поведение итераторов вставки слегка переопределено.

- Если присвоить значение элементу, ассоциированному с итератором, это значение вставляется в коллекцию, с которой связан данный итератор. Три разновидности итераторов вставки позволяют вставлять элементы в разных местах — в начале, в конце или в заданной позиции.
- Операция перехода вперед не производит никаких действий.

Рассмотрим следующий пример:

```
//stl/copy3.cpp
#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include <set>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll1;
    // Вставка элементов со значениями от 1 до 9
    // в первую коллекцию
```

```
for (int i=1; i<=9; ++i) {
    coll1.push_back(i);
}

// Копирование элементов из coll1 в coll2 с присоединением
vector<int> coll2;
copy (coll1.begin(), coll1.end(),           // Источник
      back_inserter(coll2));                // Приемник

// Копирование элементов coll1 в coll3 со вставкой в начало
// - порядок следования элементов заменяется на противоположный
deque<int> coll3;
copy (coll1.begin(), coll1.end(),           // Источник
      front_inserter(coll3));               // Приемник

// Копирование элементов coll1 в coll4
// - единственный итератор вставки, работающий
//   с ассоциативными контейнерами
set<int> coll4;
copy (coll1.begin(), coll1.end(),           // Источник
      inserter(coll4,coll4.begin()));       // Приемник
}
```

В этом примере встречаются все три разновидности итераторов вставки.

- *Конечные итераторы вставки.* Элементы вставляются в конец контейнера (то есть присоединяются) вызовом `push_back()`. Например, следующая команда присоединяет все элементы `coll1` к `coll2`:

```
copy (coll1.begin(), coll1.end(),           // Источник
      back_inserter(coll2));                // Приемник
```

Разумеется, конечные итераторы вставки могут использоваться только с контейнерами, поддерживающими функцию `push_back()`. В стандартной библиотеке C++ такими контейнерами являются векторы, деки и списки.

- *Начальные итераторы вставки.* Элементы вставляются в начало контейнера вызовом `push_front()`. Например, следующая команда вставляет все элементы `coll1` в `coll3`:

```
copy (coll1.begin(), coll1.end(),           // Источник
      front_inserter(coll3));               // Приемник
```

Помните, что данная разновидность вставки меняет порядок следования элементов. Если вставить в начало значение 1, а затем значение 2, то 1 окажется после 2.

Начальные итераторы вставки могут использоваться только с контейнерами, поддерживающими функцию `push_front()`. В стандартной библиотеке C++ такими контейнерами являются деки и списки.

- *Унифицированные итераторы вставки.* Унифицированный итератор вставки, также называемый просто *итератором вставки*, вставляет элементы непосредственно перед позицией, переданной во втором аргументе при инициализации.

Он вызывает функцию `insert()` и передает ей значение и позицию нового элемента. Вспомните, что все контейнеры, определенные в STL, поддерживают функцию `insert()`; это единственный итератор вставки в ассоциативных контейнерах.

Стоп! Но ведь выше говорилось, что задать позицию нового элемента в ассоциативном контейнере невозможно, потому что позиции элементов зависят от их значений? Все просто: в ассоциативных контейнерах переданная позиция интерпретируется как *рекомендация* для начала поиска нужной позиции. Однако если переданная позиция окажется неверной, на выполнение операции может понадобиться больше времени, чем вообще без рекомендации. На с. 291 описан пользовательский итератор вставки, более удобный при работе с ассоциативными контейнерами.

В табл. 5.1 перечислены разновидности итераторов вставки, а дополнительная информация приводится на с. 275.

Таблица 5.1. Стандартные итераторы вставки

Выражение	Разновидность итератора
<code>back_inserter</code> (контейнер)	Элементы присоединяются с конца в прежнем порядке с использованием функции <code>push_back()</code>
<code>front_inserter</code> (контейнер)	Элементы вставляются в начало в обратном порядке с использованием функции <code>push_front()</code>
<code>inserter</code> (контейнер, позиция)	Элементы вставляются в заданной позиции в прежнем порядке с использованием функции <code>insert()</code>

Потоковые итераторы

Потоковые адаптеры составляют еще одну чрезвычайно полезную разновидность итераторных адаптеров. Под этим термином понимаются итераторы, обеспечивающие чтение из потока данных и запись в поток данных¹. Иначе говоря, потоковые итераторы позволяют интерпретировать ввод с клавиатуры как коллекцию, из которой можно читать данные. Аналогично, вы можете перенаправить результаты работы алгоритма в файл или на экран.

Рассмотрим типичный пример, убедительно подтверждающий возможности STL. По сравнению с «чистой» реализацией C или C++ эта программа выполняет довольно большой объем работы всего в нескольких командах:

```
// stl/ioiter1.cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
```

¹ Потоками данных называются объекты, представляющие каналы ввода-вывода (см. главу 13).

```

int main()
{
    vector<string> coll;

    /* Загрузка слов из стандартного входного потока данных
     * - источник: все строки до конца файла (или до возникновения ошибки)
     * - приемник: coll (вставка)
     */
    copy (istream_iterator<string>(cin),           // Начало источника
          istream_iterator<string>(),               // Конец источника
          back_inserter(coll));                    // Приемник

    // Сортировка элементов
    sort (coll.begin(), coll.end());

    /* Вывод всех элементов без дубликатов
     * - источник: coll
     * - приемник: стандартный вывод (с разделением элементов
     *               символом новой строки)
     */
    unique_copy (coll.begin(), coll.end(),           // Источник
                 ostream_iterator<string>(cout, "\n")); // Приемник
}

```

Программа, фактически состоящая всего из трех команд, читает все слова из стандартного входного потока данных и выводит их в отсортированном виде. Давайте последовательно рассмотрим каждую из этих трех команд. В первой команде используются два потоковых итератора ввода:

```

copy (istream_iterator<string>(cin),
      istream_iterator<string>(),
      back_inserter(coll));

```

- Выражение `istream_iterator<string>(cin)` создает потоковый итератор для чтения из стандартного входного потока данных `cin`¹. Аргумент `string` определяет тип элементов, читаемых потоковым итератором (строковые типы рассматриваются в главе 11). Элементы читаются стандартным оператором ввода `>>`. Каждый раз, когда алгоритм пытается обработать следующий элемент, потоковый итератор ввода преобразует эту попытку в вызов:

`cin >> строка`

Оператор чтения строк обычно читает отдельное «слово», отделенное пробелом от следующего слова (см. с. 475), поэтому алгоритм вводит данные по словам.

- Выражение `istream_iterator<string>()` вызывает конструктор по умолчанию и создает *конечный потоковый итератор*. Он представляет поток данных, дальнейшее чтение из которого невозможно.

¹ В старых системах во втором параметре шаблона должен присутствовать тип `ptrdiff_t` (см. с. 281).

Как обычно, алгоритм `copy()` продолжает работать до тех пор, пока первый (наращиваемый) аргумент не сравняется со вторым аргументом. Конечный потоковый итератор определяет *конец обрабатываемого интервала*; следовательно, алгоритм будет читать все строки из `cin` до тех пор, пока дальнейшее чтение станет невозможным (из-за достижения конца потока данных или из-за ошибки).

Подведем итог: источником данных для алгоритма `copy()` являются «все слова из `cin`». Прочитанные слова копируются в `coll` конечным итератором вставки.

Алгоритм `sort()` сортирует все содержимое коллекции:

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

Наконец, следующая команда копирует все элементы коллекции в приемник `cout`:

```
unique_copy (coll.begin(), coll.end(),
            ostream_iterator<string>(cout, "\n"));
```

В процессе копирования алгоритм `unique_copy` исключает из копируемых данных дубликаты. Представленное ниже выражение создает потоковый итератор вывода, который записывает `string` в выходной поток данных `cout`, вызывая оператор `<<` для каждого элемента:

```
ostream_iterator<string>(cout, "\n"));
```

Второй аргумент определяет разделитель, выводимый между элементами. Передавать этот аргумент не обязательно. В нашем примере разделителем является символ новой строки, поэтому каждый элемент будет выводиться в отдельной строке.

Все компоненты программы реализованы в виде шаблонов, поэтому программа легко адаптируется для сортировки других типов данных (например, целых чисел или более сложных объектов). На с. 281 эта тема рассматривается более подробно и поясняется дополнительными примерами использования потоковых итераторов ввода-вывода.

В нашем примере для сортировки всех слов в стандартном входном потоке данных понадобились одно объявление и три команды. Однако аналогичного результата можно добиться всего одним объявлением и одной командой. Пример приведен на с. 232.

Обратные итераторы

Третей разновидностью итераторных адаптеров STL являются обратные итераторы. Эти итераторы работают в противоположном направлении: вызов оператора `++` на внутреннем уровне преобразуется в вызов оператора `--`, и наоборот. Все контейнеры поддерживают создание обратных итераторов функциями `rbegin()` и `rend()`. Рассмотрим следующий пример:

```
// stl/riter1.cpp
#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;

int main()
{
    vector<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // Вывод всех элементов в обратном порядке
    copy (coll.rbegin(), coll.rend(),           // Источник
          ostream_iterator<int>(cout, " "));   // Приемник
    cout << endl;
}
```

Выражение `coll.rbegin()` возвращает обратный итератор для коллекции `coll`. Полученный итератор может использоваться, чтобы начать обратный перебор элементов коллекции, поскольку он устанавливается на последний элемент коллекции. Таким образом, выражение `*coll.rbegin()` возвращает значение последнего элемента.

Соответственно выражение `coll.rend()` возвращает обратный итератор для коллекции `coll`, который может использоваться для завершения перебора в обратном направлении. По аналогии с обычными интервалами он устанавливается за концом интервала, но в противоположном направлении — то есть перед первым элементом коллекции.

Значение выражения `*coll.rend()` не определено, как, впрочем, и значение `*coll.end()`. Никогда не используйте операторы `*` и `->` с позициями, не представляющими реальных элементов коллекции.

Основное достоинство обратных итераторов заключается в том, что все алгоритмы получают возможность работать в обратном направлении без модификации кода. Переход к следующему элементу оператором `++` преобразуется в переход к предыдущему элементу оператором `--`. В нашем примере алгоритм `copy()` перебирает все элементы `coll` от последнего к первому, поэтому результат работы программы выглядит так:

9 8 7 6 5 4 3 2 1

«Нормальные» итераторы можно переключать в режим обратного перебора и наоборот. Однако следует помнить, что при этом изменяется текущий элемент, связанный с итератором. Дополнительная информация об обратных итераторах приводится на с. 270.

Модифицирующие алгоритмы

Некоторые алгоритмы изменяют содержимое своих приемных интервалов. В частности, возможно удаление элементов. При этом необходимо учитывать некоторые обстоятельства, которые будут описаны в этом разделе. Происходящее порой

выглядит довольно странно, но это неизбежное следствие архитектуры STL, основанной на гибком разделении контейнеров и алгоритмов.

«Удаление» элементов

Алгоритм `remove()` удаляет элементы из заданного интервала. Но если вызвать его для всех элементов контейнера, происходит нечто неожиданное. Пример:

```
// stl/remove1.cpp
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // Вставка элементов со значениями от 6 до 1 и от 1 до 6
    for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
    }

    // Вывод всех элементов коллекции
    cout << "pre: ";
    copy (coll.begin(), coll.end(),           // Источник
          ostream_iterator<int>(cout, " ")); // Приемник
    cout << endl;

    // Удаление всех элементов со значением 3
    remove (coll.begin(), coll.end(),           // Интервал
            3);                                // Значение

    // Вывод всех элементов коллекции
    cout << "post: ";
    copy (coll.begin(), coll.end(),           // Источник
          ostream_iterator<int>(cout, " ")); // Приемник
    cout << endl;
}
```

При поверхностном знакомстве с предметом напрашивается предположение, что эта программа удаляет из коллекции все элементы со значением 3. Но на самом деле результат выглядит так:

```
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
```

Алгоритм `remove()` не изменил количество элементов в коллекции, для которой он вызывался. Функция `end()` возвращает прежнее значение, а функция

`size()` — прежнее количество элементов. Тем не менее кое-что все же изменилось: порядок следования элементов стал таким, каким он должен стать после удаления. На место элементов со значением 3 были записаны следующие элементы (рис. 5.7). Прежние элементы в конце коллекции, не перезаписанные алгоритмом, остались без изменений. На логическом уровне эти элементы уже не принадлежат коллекции.

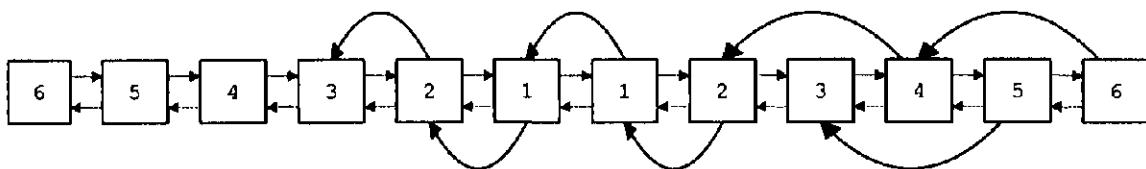


Рис. 5.7. Принцип действия алгоритма `remove()`

Однако алгоритм возвращает итератор, указывающий на новый конец коллекции. Это значение позволяет прочитать полученный интервал, уменьшить размер коллекции или узнать количество удаленных элементов. Рассмотрим слегка измененную версию примера:

```
// stl/remove2.cpp
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // Вставка элементов со значениями от 6 до 1 и от 1 до 6
    for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
    }

    // Вывод всех элементов коллекции
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout, " "));
    cout << endl;

    // Удаление всех элементов со значением 3
    // - сохранение нового конца коллекции
    list<int>::iterator end = remove (coll.begin(), coll.end(),
                                       3);

    // Вывод элементов полученной коллекции
    copy (coll.begin(), end,
          ostream_iterator<int>(cout, " "));
    cout << endl;
}
```

```

// Вывод количества "удаленных" элементов
cout << "number of removed elements: "
    << distance(end, coll.end()) << endl;

// Стирание "удаленных" элементов
coll.erase (end, coll.end());

// Вывод всех элементов модифицированной коллекции
copy (coll.begin(), coll.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;
}

```

В новой версии программы значение, возвращаемое алгоритмом `remove()`, присваивается итератору `end`:

```
list<int>::iterator end = remove (coll.begin(), coll.end(),
                                    3);
```

Итератор `end` определяет «логический» конец модифицированной коллекции после «удаления» элементов. Полученное значение определяет конечную границу интервала при последующих операциях:

```
copy (coll.begin(), end,
      ostream_iterator<int>(cout, " "));
```

Другой способ основан на обработке количества «удаленных» элементов, вычисляемого как расстояние между «логическим» и «физическими» концами коллекции:

```
cout << "number of removed elements: "
    << distance(end, coll.end()) << endl;
```

В этом фрагменте используется специальная вспомогательная функция `distance()`, возвращающая расстояние между двумя итераторами. При работе с итераторами произвольного доступа расстояние вычисляется простым вычитанием оператором `-`, однако в нашем примере используется список, который поддерживает только двунаправленные итераторы. Дополнительная информация о функции `distance()` приведена на с. 267¹.

Чтобы «удаленные» элементы были действительно удалены из коллекции, следует вызвать соответствующую функцию контейнера. Для таких целей контейнеры поддерживают функцию `erase()`, которая удаляет все элементы из интервала, определяемого переданными аргументами:

```
coll.erase (end, coll.end());
```

Результат работы программы:

```
6 5 4 3 2 1 1 2 3 4 5 6
6 5 4 2 1 1 2 4 5 6
```

¹ Определение `distance()` было изменено, поэтому в старых версиях STL необходимо включать файл `distance.hpp` (см. с. 268).

```
number of removed elements: 2
6 5 4 2 1 1 2 4 5 6 5 6
```

Если вы предпочитаете удалять элементы одной командой, воспользуйтесь следующей конструкцией:

```
coll.erase(remove(coll.begin(), coll.end(),
                 3),
            coll.end());
```

Почему сами алгоритмы не вызывают функцию `erase()`? Ответ на этот вопрос снова напоминает о том, что гибкость STL не дается даром. STL отделяет структуры данных от алгоритмов, используя итераторы в качестве интерфейсов. Но как отмечалось выше, итератор является абстракцией для представления позиции в контейнере. В общем случае итератор просто *не знает* свой контейнер. Таким образом, алгоритм, работающий с элементами контейнера через итератор, не может вызывать функции контейнера.

Подобная архитектура имеет важные последствия, поскольку благодаря ей алгоритмы работают с абстрактными интервалами, а не только со «всеми элементами контейнера». Например, интервал может состоять из подмножества элементов коллекции. Более того, некоторые контейнеры (например, обычные массивы) не поддерживают функцию `erase()`. Следовательно, стремление сохранить максимальную гибкость алгоритмов не позволяет требовать, чтобы итераторы располагали информацией о своих контейнерах.

Наконец, необходимость в уничтожении «удаленных» элементов возникает не так уж часто. Часто проще использовать новый «логический» конец контейнера вместо «физического», в частности, при последующем вызове всех алгоритмов.

Модифицирующие алгоритмы и ассоциативные контейнеры

При попытке использования модифицирующих алгоритмов (то есть алгоритмов, которые удаляют элементы, изменяют порядок их следования или их значения) возникает другая проблема: ассоциативные контейнеры не могут быть приемниками. Причина проста: работая с ассоциативным контейнером, модифицирующий алгоритм может изменить значения или позиции элементов, что приведет к нарушению порядка их сортировки. Тем самым будет нарушено основное правило, согласно которому элементы ассоциативных контейнеров всегда сортируются автоматически по заданному критерию. Поэтому для сохранения упорядоченности все итераторы ассоциативных контейнеров объявляются итераторами константных значений (или ключей). Следовательно, попытки модификации элементов ассоциативного контейнера через итератор вызывают ошибки на стадии компиляции¹.

¹ К сожалению, в некоторых системах обработка ошибок реализована крайне плохо. Вы понимаете, что в программе что-то явно не так, но причины происходящего остаются неизвестными. Некоторые компиляторы даже не отображают фрагмент исходного текста, в котором обнаружена ошибка. Вероятно, в будущем ситуация изменится.

Учите, что это обстоятельство не позволяет вызывать алгоритмы удаления для ассоциативных контейнеров, потому что эти алгоритмы косвенно модифицируют элементы. Значения «удаленных» элементов перезаписываются следующими «неудаленными» элементами.

Возникает вопрос: как же удалить элементы из ассоциативного контейнера? Ответ прост: воспользуйтесь собственными функциями контейнера! В каждом ассоциативном контейнере предусмотрены функции удаления элементов. Например, элементы можно удалить функцией `erase()`:

```
// stl/remove3.cpp
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;

int main()
{
    set<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.insert(i);
    }

    // Вывод всех элементов коллекции
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout, " "));
    cout << endl;

    /* Удаление всех элементов со значением 3
     * - алгоритм remove() не работает
     * - используем функцию erase()
     */
    int num = coll.erase(3);

    // Вывод количества удаленных элементов
    cout << "number of removed elements: " << num << endl;

    // Вывод всех элементов модифицированной коллекции
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout, " "));
    cout << endl;
}
```

Контейнеры поддерживают несколько версий функции `erase()`. Только та версия, единственным аргументом которой является значение удаляемого элемента (или элементов), возвращает количество удаленных элементов (см. с. 248). Разумеется, если дубликаты в коллекции запрещены (как в множествах и отображениях), возвращаемое значение функции может быть равно только 0 или 1.

Результат работы программы выглядит так:

```
1 2 3 4 5 6 7 8 9  
number of removed elements: 1  
1 2 4 5 6 7 8 9
```

Алгоритмы и функции классов

Если можно использовать алгоритм, это еще не значит, что его нужно использовать. Возможно, контейнерный класс содержит функции, которые работают гораздо эффективнее.

Хорошим примером служит вызов алгоритма `remove()` для элементов списка. Алгоритм не знает, что он работает со списком, поэтому он делает то же, что в любом другом контейнере, — переупорядочивает элементы, изменяя их значения. Например, при удалении первого элемента каждому элементу присваивается значение элемента, следующего после него. Такой подход игнорирует основное достоинство списков — возможность выполнения вставки, перемещения и удаления элементов посредством модификации ссылок, а не элементов.

Для повышения эффективности операций в списках определены специальные функции для всех модифицирующих алгоритмов. Всегда используйте их вместо алгоритмов. Более того, как показывает следующий пример, эти функции действительно исключают удаляемые элементы:

```
// stl/remove4.cpp  
#include <iostream>  
#include <list>  
#include <algorithm>  
using namespace std;  
  
int main()  
{  
    list<int> coll;  
  
    // Вставка элементов со значениями от 6 до 1 и от 1 до 6  
    for (int i=1; i<=6; ++i) {  
        coll.push_front(i);  
        coll.push_back(i);  
    }  
  
    // Удаление всех элементов со значением 3  
    // - незэффективно  
    coll.erase(remove(coll.begin(), coll.end(),  
                    3),  
               coll.end());  
  
    // Удаление всех элементов со значением 4  
    // - эффективно  
    coll.remove(4);  
}
```

Если вы стремитесь обеспечить хорошее быстродействие, всегда используйте функции классов вместо алгоритмов. Есть только одна проблема — вы должны заранее знать, что некоторый контейнер содержит функцию, обладающую более высоким быстродействием. При вызове алгоритма `remove()` не последует ни предупреждения, ни сообщения об ошибке. С другой стороны, если выбрать функцию, то при переходе на другой тип контейнера вам придется вносить изменения в программу. В справочнике алгоритмов (см. главу 9) указано, когда лучше использовать функцию классов, превосходящую алгоритм по быстродействию.

Унифицированные пользовательские функции

Библиотека STL имеет расширяемую архитектуру. Это означает, что программист может создавать собственные функции и алгоритмы для обработки коллекций. Конечно, такие операции тоже должны быть унифицированными, но при объявлении в них действительного итератора необходимо указывать тип контейнера. Чтобы упростить написание унифицированных функций, в каждом контейнерном классе присутствуют внутренние определения вспомогательных типов. Рассмотрим следующий пример:

```
// stl/print.hpp
#include <iostream>

/* PRINT_ELEMENTS()
 * - вывод необязательной строки C optcstr, после которой
 * - выводятся все элементы коллекции coll.
 * - разделенные пробелами.
 */
template <class T>
inline void PRINT_ELEMENTS (const T& coll, const char* optcstr="")
{
    typename T::const_iterator pos;

    std::cout << optcstr;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}
```

В этом примере определяется унифицированная функция для вывода необязательной строки, за которой выводятся все элементы переданного контейнера. В следующем объявлении переменная `pos` объявляется с типом итератора для контейнера переданного типа:

```
typename T::const_iterator pos;
```

Ключевое слово `typename` указывает, что `const_iterator` — тип, а не значение типа `T` (см. с. 27).

Помимо типов `iterator` и `const_iterator` контейнеры предоставляют и другие типы, упрощающие написание унифицированных функций. В частности, они предоставляют тип элементов для выполнения операций с временными копиями элементов. За подробностями обращайтесь на с. 289.

Необязательный второй аргумент функции `PRINT_ELEMENTS` представляет собой строковый префикс, выводимый перед содержимым контейнера. Так, функция `PRINT_ELEMENTS()` позволяет снабжать выходные данные комментариями:

```
PRINT_ELEMENTS(coll, "all elements: ");
```

Эта функция будет еще часто использоваться в книге для вывода всех элементов контейнера одной простой командой.

Передача функций алгоритмам в качестве аргументов

Некоторым алгоритмам могут передаваться пользовательские функции, которые вызываются в процессе внутренней работы алгоритма. Такая возможность делает алгоритмы еще более гибкими и мощными.

Примеры передачи функций в аргументах алгоритмов

Передачу функции проще всего рассматривать на примере алгоритма `for_each()`, вызывающего пользовательскую функцию для каждого элемента в заданном интервале.

```
// stl/foreach.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Функция выводит переданный аргумент
void print (int elem)
{
    cout << elem << ' ';
}

int main()
{
    vector<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
```

```

        coll.push_back(i);
    }

// Вывод всех элементов
for_each (coll.begin(), coll.end(),      // Интервал
          print);                         // Операция
cout << endl;
}

```

Алгоритм `for_each()` вызывает функцию `print()` для каждого элемента в интервале `[coll.begin(),coll.end()]`. Результат выполнения программы выглядит так:

```
1 2 3 4 5 6 7 8 9
```

Существует несколько вариантов применения пользовательских функций алгоритмами; в одних случаях передача функция обязательна, в других — нет. В частности, пользовательская функция может определять критерий поиска, критерий сортировки или преобразование, применяемое при копировании элементов из одной коллекции в другую.

Другой пример:

```

// stl/transform1.cpp
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
#include "print.hpp"

int square (int value)
{
    return value*value;
}

int main()
{
    std::set<int>    coll1;
    std::vector<int>  coll2;

// Вставка элементов со значениями от 1 до 9 в coll1
for (int i=1; i<=9; ++i) {
    coll1.insert(i);
}
PRINT_ELEMENTS(coll1,"initialized: ");

// Преобразование каждого элемента при копировании из coll1 в coll2
// - square transformed values
std::transform (coll1.begin(),coll1.end(),      // Источник
               std::back_inserter(coll2),      // Приемник
               square);                      // Операция

PRINT_ELEMENTS(coll2,"squared:      ");
}

```

В приведенном примере функция `square()` возводит в квадрат каждый элемент `coll1` при его копировании в `coll2` (рис. 5.8). Результат выполнения программы выглядит так:

```
initialized: 1 2 3 4 5 6 7 8 9
squared:     1 4 9 16 25 36 49 64 81
```

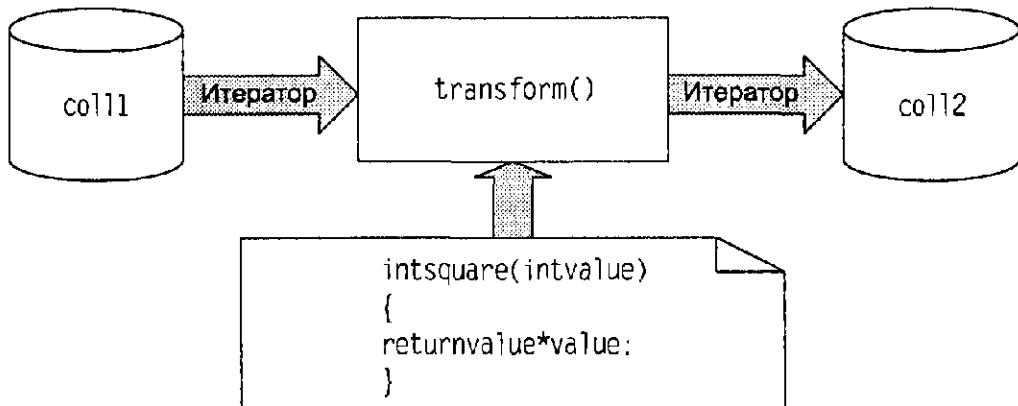


Рис. 5.8. Работа алгоритма `transform()`

Предикаты

Особую разновидность вспомогательных функций, используемых алгоритмами, составляют *предикаты*. Предикатом называется функция, которая возвращает логическое значение. С помощью предикатов часто определяют критерии сортировки или поиска. В зависимости от способа применения предикаты делятся на унарные и бинарные. Учтите, что не любая унарная или бинарная функция, возвращающая логическую величину, является действительным предикатом. STL требует, чтобы при неизменности входных данных предикат всегда давал постоянный результат. Тем самым из категории предикатов исключаются функции, внутреннее состояние которых изменяется в процессе вызова. За подробностями обращайтесь на с. 303.

Унарные предикаты

Унарный предикат проверяет некоторое свойство одного аргумента. Типичный пример — функция, используемая в качестве критерия поиска первого простого числа:

```
// stl/prime1.cpp
#include <iostream>
#include <list>
#include <algorithm>
#include <cstdlib>      // для abs()
using namespace std;

// Предикат проверяет, является ли целое число простым
bool isPrime (int number)
{
```

```

// Знак числа игнорируется
number = abs(number);

// 0 и 1 не являются простыми числами
if (number == 0 || number == 1) {
    return false;
}

// Поиск множителя, на который число делится без остатка
int divisor;
for (divisor = number/2; number%divisor != 0; --divisor) {
    ;
}
// Если не найдено ни одного множителя, большего 1,
// проверяемое число является простым.
return divisor == 1;
}

int main()
{
    list<int> coll;

    // Вставка элементов со значениями от 24 до 30
    for (int i=24; i<=30; ++i) {
        coll.push_back(i);
    }

    // Поиск простого числа
    list<int>::iterator pos;
    pos = find_if (coll.begin(), coll.end(),           // Интервал
                   isPrime);                         // Предикат
    if (pos != coll.end()) {
        // Найдено простое число
        cout << *pos << " is first prime number found" << endl;
    }
    else {
        // Простые числа не найдены
        cout << "no prime number found" << endl;
    }
}

```

В приведенном примере алгоритм `find_if()` ищет в заданном интервале первый элемент, для которого передаваемый унарный предикат возвращает `true`. В качестве предиката передается функция `isPrime()`, которая проверяет, является ли число простым. В итоге алгоритм возвращает первое простое число в заданном интервале. Если алгоритм не находит ни одного элемента, удовлетворяющего предикату, возвращается конец интервала (второй аргумент). Это условие проверяется после вызова. Коллекция из нашего примера содержит простое число между 24 и 30, поэтому результат выполнения программы выглядит так:

29 is first prime number found

Бинарные предикаты

Бинарные предикаты обычно сравнивают некоторое свойство двух аргументов. Например, чтобы отсортировать элементы по нестандартному критерию, программист передает алгоритму простую предикатную функцию. Это может понадобиться, если, например, элементы не могут корректно работать с оператором < или сортировка должна выполняться по нестандартному критерию.

В следующем примере множество с информацией о группе людей сортируется по именам и фамилиям:

```
// stl/sort1.cpp
#include <iostream>
#include <string>
#include <deque>
#include <algorithm>
using namespace std;

class Person {
public:
    string firstname() const;
    string lastname() const;
    ...
};

/* Бинарный предикат:
 * - сравнивает два объекта Person
 */
bool personSortCriterion (const Person& p1, const Person& p2)
{
    /* Первый объект Person меньше второго,
     * - если фамилия в первом объекте меньше фамилии во втором объекте;
     * - или если фамилии равны, а имя в первом объекте меньше.
     */
    return p1.lastname()<p2.lastname() ||
        (!(p2.lastname()<p1.lastname()) &&
         p1.firstname()<p2.firstname());
}

int main()
{
    deque<Person> coll;
    ...
    sort(coll.begin(),coll.end(),      // Интервал
          personSortCriterion);       // Критерий сортировки
    ...
}
```

Критерий сортировки также может определяться в виде объекта функции. У подобной реализации есть свои преимущества — критерий становится типом,

который может использоваться, например, при объявлении множества, сортирующего свои элементы по этому критерию. Реализация критерия сортировки в виде объекта функции описана на с. 296.

Объекты функций

Функциональные аргументы, передаваемые алгоритмам, не обязаны быть функциями. Они могут быть объектами, которые ведут себя как функции. Такие объекты называются *объектами функций*, или *функциорами*. Иногда объект функции справляется с ситуацией, в которой обычная функция не работает. Объекты функций часто используются в STL, причем некоторые являются очень полезными.

Понятие объекта функции

Объекты функций — еще один пример унифицированного программирования и концепции чистой абстракции. В соответствии с этой концепцией все, что *ведет себя как функция, является функцией*. Следовательно, если определить объект, который ведет себя как функция, он может использоваться в качестве функции.

Но как понимать фразу «ведет себя как функция»? Ответ: это означает возможность вызова с круглыми скобками и передачей аргументов. Пример:

```
function(arg1,arg2); // Вызов функции
```

Следовательно, если мы хотим, чтобы объект вел себя подобным образом, необходимо обеспечить возможность его «вызыва» в программе с круглыми скобками и передачей аргументов. Да, такое вполне возможно (в C++ вообще редко встречается что-нибудь невозможное). От вас лишь потребуется определить оператор () с соответствующими типами параметров:

```
class X {  
public:  
    // Определение оператора "вызов функции"  
    возвращаемое_значение operator() (аргументы) const;  
    ...  
};
```

Теперь объекты этого класса ведут себя как функции и могут вызываться в программе:

```
X fo;  
...  
fo(arg1,arg2); // Вызов оператора () для объекта функции fo
```

Такой вызов эквивалентен следующему:

```
fo.operator()(arg1,arg2); // Вызов оператора () для объекта функции fo
```

Ниже приведен более полный пример — версия рассмотренной ранее программы с обычной функцией (см. с. 129), переработанная для использования объекта функции:

```
// stl/foreach2.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Простой объект функции для вывода передаваемого аргумента
class PrintInt {
public:
    void operator() (int elem) const {
        cout << elem << ' ';
    }
};

int main()
{
    vector<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // Вывод всех элементов
    for_each (coll.begin(), coll.end(), // Интервал
              PrintInt());           // Операция
    cout << endl;
}
```

Класс `PrintInt` определяет объект, для которого вызывается оператор `()` с аргументом `int`. Выражение `PrintInt()` в следующей команде создает временный объект этого класса, передаваемый алгоритму `for_each()` в качестве аргумента:

```
for_each (coll.begin(), coll.end(),
          PrintInt());
```

Алгоритм `for_each()` определяется примерно так:

```
namespace std {
    template <class Iterator, class Operation>
    Operation for_each (Iterator act, Iterator end, Operation op)
    {
        while (act != end) { // Пока не достигнут конец интервала
            op(*act);       // - вызвать op() для текущего элемента
            ++act;          // - переместить итератор к следующему
        }                  // элементу
        return op;
    }
}
```

Алгоритм `for_each()` использует временный объект функции `op`, чтобы для каждого элемента делать вызов `op(*act)`. Если третий параметр является обычной функцией, она просто вызывается с аргументом `*act`. Если третий параметр является объектом функции, вызывается оператор `()` объекта функции `op` с аргументом `*act`. Таким образом, в приведенной программе алгоритм `for_each()` вызывает операторную функцию

```
PrintInt::operator()(*act)
```

На первый взгляд непонятно, зачем это нужно. Возникает ощущение, что объекты функций — конструкция странная, непонятная, а то и вовсе бессмысличная. Действительно, объекты функций усложняют программу. Но они способны на большее, чем функции, обладая рядом несомненных достоинств.

- *Объект функции — это «умная функция».* Объекты, которые ведут себя как указатели, называются «умными указателями». По аналогии объекты, которые ведут себя как функции, можно назвать «умными функциями», поскольку их возможности не ограничиваются вызовом оператора `()`. Объекты функций могут представлять другие функции и иметь другие атрибуты, а это означает, что объекты функций обладают состоянием. Одна и та же функция, представленная объектом функции, в разные моменты времени может находиться в разных состояниях. У обычных функций такая возможность отсутствует. Из этого следует одно из достоинств объектов функций — возможность их инициализации на стадии выполнения перед вызовом/использованием.
- *Каждому объекту функции соответствует свой тип.* Обычные функции обладают разными типами только при различиях в их сигнатурах. С другой стороны, объекты функций могут иметь разные типы даже при полном совпадении сигнатур. Это открывает новые возможности для унифицированного программирования с применением шаблонов, потому что функциональное поведение может передаваться как параметр шаблона. Например, это позволяет контейнерам разных типов использовать единственный объект функции в качестве критерия сортировки. Тем самым предотвращается возможное присваивание, слияние и сравнение коллекций, использующих разные критерии сортировки. Вы даже можете спроектировать иерархию объектов функций, например, для определения нескольких специализированных версий одного общего критерия.
- *Объекты функций обычно работают быстрее функций.* Шаблоны обычно лучше оптимизируются, поскольку на стадии компиляции доступно больше информации. Следовательно, передача объекта функции вместо обычной функции часто повышает быстродействие программы.

В оставшейся части этого подраздела приводятся несколько примеров, доказывающих, что объекты функций действительно «умнее» обычных функций. Дополнительные примеры и подробности приводятся в главе 8, полностью посвященной объектам функций. В частности, вы узнаете, насколько полезной бывает передача функционального поведения в качестве параметра шаблона.

Допустим, требуется увеличить значения всех элементов коллекции на некоторую величину. Если приращение известно на стадии компиляции, можно воспользоваться обычной функцией:

```
void add10 (int& elem)
{
    elem += 10;
}

void f1()
{
    vector<int> coll;
    ...
    for_each(coll.begin(), coll.end(),      // Интервал
              add10);                      // Операция
}
```

Если возможны разные приращения, известные на стадии компиляции, функция оформляется в виде шаблона:

```
template <int theValue>
void add (int& elem)
{
    elem += theValue;
}

void f1()
{
    vector<int> coll;
    ...
    for_each(coll.begin(), coll.end(),      // Интервал
              add<10>);                  // Операция
}
```

Но если приращение определяется только во время выполнения программы, ситуация усложняется. Значение должно быть передано функции перед ее вызовом. Обычно в таких случаях применяется глобальная переменная, используемая как функцией, вызывающей алгоритм, так и функцией, вызываемой алгоритмом для суммирования. Программа получается неряшливой и запутанной.

Если функция вызывается дважды с двумя разными приращениями, оба из которых становятся известными только во время выполнения, такая задача в принципе не решается одной обычной функцией. Приходится либо передавать функции дополнительный признак, либо писать две разные функции. Вам когда-нибудь приходилось копировать определение функции, потому что ее состояние хранилось в статической переменной, а вам требовалась та же функция, но с другим состоянием? Наша проблема в точности аналогична этой.

Объект функции позволяет написать «умную» функцию, которая ведет себя нужным образом. Поскольку объект обладает состоянием, мы можем инициализировать его правильным приращением. Ниже приведен полный код примера¹:

```
// stl/add1.cpp
#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

// Объект функции прибавляет к значению элемента приращение,
// заданное при его инициализации
class AddValue {
private:
    int theValue; // Приращение
public:
    // Конструктор инициализирует приращение
    AddValue(int v) : theValue(v) {
    }

    // Суммирование выполняется "вызовом функции" для элемента
    void operator() (int& elem) const {
        elem += theValue;
    }
};

int main()
{
    list<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    PRINT_ELEMENTS(coll, "initialized: ");

    // Прибавить к каждому элементу 10
    for_each (coll.begin(), coll.end(), // Интервал
              AddValue(10)); // Операция

    PRINT_ELEMENTS(coll, "after adding 10: ");

    // Прибавить к каждому элементу значение первого элемента
}
```

¹ Вспомогательная функция PRINT_ELEMENTS() представлена на с. 128.

```
for_each (coll.begin(), coll.end(),    // Интервал
          AddValue(*coll.begin()));    // Операция

PRINT_ELEMENTS(coll, "after adding first element: ");
}
```

После инициализации коллекция содержит числа от 1 до 9:

initialized: 1 2 3 4 5 6 7 8 9

Первый вызов `for_each()` увеличивает каждый элемент на 10:

```
for_each (coll.begin(), coll.end(),    // Интервал
          AddValue(10));           // Операция
```

Выражение `AddValue(10)` создает объект типа `AddValue` и инициализирует его значением 10. Конструктор `AddValue` сохраняет это значение в переменной `theValue`. Во время работы `for_each()` для каждого элемента `coll` вызывается оператор `()`. Еще раз стоит уточнить, что речь идет о вызове оператора `()` для переданного временного объекта функции типа `AddValue`. Сам элемент передается в аргументе. Объект функции увеличивает каждый элемент на 10. В результате содержимое коллекции выглядит так:

after adding 10: 11 12 13 14 15 16 17 18 19

Второй вызов `for_each()` тем же способом прибавляет значение первого элемента к каждому элементу коллекции. При этом временный объект функции `AddValue` инициализируется первым элементом коллекции:

```
AddValue(*coll.begin())
```

На этот раз результат выглядит так:

after adding first element: 22 23 24 25 26 27 28 29 30

На с. 334 приведена измененная версия этого примера, в которой объект функции `AddValue` оформлен в виде шаблона для типа прибавляемого значения.

Применение этой методики решает проблему одновременного существования двух состояний одной функции. Например, можно просто объявить два объекта функции и использовать их независимо друг от друга:

```
AddValue addx(x);      // Объект функции, прибавляющий значение x
AddValue addy(y);      // Объект функции, прибавляющий значение y

for_each (coll.begin(), coll.end(),    // Прибавление значения x
          addx);                  // к каждому элементу
...
for_each (coll.begin(), coll.end(),    // Прибавление значения y
          addy);                  // к каждому элементу
...
for_each (coll.begin(), coll.end(),    // Прибавление значения x
          addx);                  // к каждому элементу
```

Аналогичным способом определяются другие функции для получения или изменения состояния объекта функции на протяжении его жизненного цикла. Пример приведен на с. 335.

Учтите, что для некоторых алгоритмов стандартная библиотека C++ не определяет количество вызовов объекта для каждого элемента, поэтому нельзя исключать, что элементу будут передаваться разные экземпляры объекта функции. Это может привести к нежелательным последствиям, в частности, если объекты функций используются как предикаты. Проблема рассматривается на с. 303.

Стандартные объекты функций

Стандартная библиотека C++ содержит несколько объектов функций для выполнения базовых операций. Иногда эти объекты избавляют программиста от необходимости создавать собственные объекты функций. Типичным примером служит объект функции, используемый как критерий сортировки. По умолчанию сортировка с применением оператора `<` осуществляется с критерием `less<>`. Рассмотрим следующее объявление:

```
set<int> coll;
```

Это объявление расширяется до¹

```
set<int, less<int>> coll; // Сортировка элементов оператором <
```

Так же просто обеспечивается сортировка элементов в обратном порядке²:

```
set<int, greater<int>> coll; // Сортировка элементов оператором >
```

Многие стандартные объекты функций предназначены для операций с числовыми данными. Например, следующая команда меняет знак всех элементов коллекции:

```
transform (coll.begin(), coll.end(),           // Источник
          coll.begin(),                   // Приемник
          negate<int>());               // Операция
```

Выражение `negate<int>()` создает объект функции стандартного шаблонного класса `negate`. Этот объект просто возвращает элемент типа `int`, для которого он был вызван, с противоположным знаком. Алгоритм `transform()` использует эту операцию для преобразования всех элементов первой коллекции во вторую коллекцию. Если источник и приемник совпадают (как в нашем случае), возвращаемые элементы с измененным знаком записываются на место исходных. Таким образом, приведенная выше команда меняет знак всех элементов коллекции.

¹ В системах, не поддерживающих аргументы шаблонов по умолчанию, обычно приходится использовать вторую форму.

² Обратите внимание на пробел между символами `>`. Последовательность `>` воспринимается компилятором как оператор сдвига, что приводит к синтаксической ошибке.

Возведение всех элементов коллекции в квадрат выполняется аналогично:

```
transform (coll.begin(), coll.end(),           // Первый источник
          coll.begin(),                   // Второй источник
          coll.begin(),                   // Приемник
          multiplies<int>());           // Операция
```

На этот раз другая форма алгоритма `transform()` объединяет элементы двух коллекций, применяя к ним заданную операцию, и записывает результат в третью коллекцию. В данном примере во всех трех случаях используется одна и та же коллекция, поэтому каждый элемент умножается сам на себя, а результат записывается в исходную позицию¹.

Специальные функциональные адаптеры позволяют объединять стандартные объекты функций с другими значениями или определять особые ситуации. Пример:

```
// stl/f01.cpp
#include <iostream>
#include <set>
#include <deque>
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
    set<int,greater<int> > coll1;
    deque<int> coll2;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll1.insert(i);
    }

    PRINT_ELEMENTS(coll1,"initialized: ");

    // Преобразование всех элементов coll2 умножением на 10
    transform (coll1.begin(),coll1.end(),           // Источник
               back_inserter(coll2),                // Приемник
               bind2nd(multiplies<int>(),10));     // Операция

    PRINT_ELEMENTS(coll2,"transformed: ");

    // Замена значения, равного 70, на 42
    replace_if (coll2.begin(),coll2.end(),           // Интервал
                bind2nd(equal_to<int>(),70),         // Критерий замены
                42);                                // Новое значение
```

¹ В предыдущих версиях STL объект функции, выполнявший умножение, назывался `times`. Переименование произошло из-за конфликта имен с функциями стандартов операционных систем (X/Open, POSIX), а также потому, что имя `multiplies` более понятно.

```

PRINT_ELEMENTS(col12,"replaced:    ");

// Удаление всех элементов со значениями, меньшими 50
col12.erase(remove_if(col12.begin(),col12.end(), // Интервал
                      bind2nd(less<int>(),50)), // Критерий удаления
            col12.end()); 

PRINT_ELEMENTS(col12,"removed:    ");
}

```

Следующая команда умножает каждый элемент `col11` на 10 и вставляет его в `col12`:

```

transform (col11.begin(),col11.end(),           // Источник
          back_inserter(col12),                  // Приемник
          bind2nd(multiplies<int>(),10));     // Операция

```

Функциональный адаптер `bind2nd` обеспечивает вызов `multiplies<int>` для каждого элемента исходной коллекции (первый аргумент) и множителя 10 (второй аргумент).

При вызове `bind2nd()` происходит следующее: в четвертом аргументе алгоритм `transform()` рассчитывает получить операцию, которая вызывается с одним аргументом (элементом, с которым она выполняется). В нашем примере аргумент умножается на 10. Следовательно, мы должны объединить операцию с двумя аргументами и постоянную величину, которая всегда будет использоваться вместо второго аргумента; в результате будет получена нужная операция с одним аргументом. Именно это и делает адаптер `bind2nd()`. Он сохраняет операцию и второй аргумент в своих внутренних данных. Когда алгоритм вызывает адаптер `bind2nd` для конкретного элемента, вызывается операция с переданным элементом (первый аргумент) и внутренним значением (второй аргумент), а возвращается полученный результат.

В аналогичном фрагменте выражение `bind2nd(equal_to<int>(),70)` определяет критерий отбора элементов, заменяемых значением 42:

```

replace_if (col12.begin(),col12.end(),
            bind2nd(equal_to<int>(),70),
            42);

```

Адаптер `bind2nd` вызывает бинарный предикат `equal_to` со вторым аргументом, равным 70. Тем самым определяется унарный предикат для элементов обрабатываемой коллекции.

Последняя команда действует аналогично. Выражение `bind2nd(less<int>(),50)` определяет элементы, которые должны быть удалены из коллекции. Команда удаляет все элементы со значением меньше 50. Результат выполнения программы выглядит так:

```

initialized: 9 8 7 6 5 4 3 2 1
transformed: 90 80 70 60 50 40 30 20 10
replaced:   90 80 42 60 50 40 30 20 10
removed:    90 80 60 50

```

Подобный подход к программированию приводит к *функциональной композиции*. Интересная подробность: все объекты функций обычно объявляются подставляемыми (*inline*). Таким образом, несмотря на абстрактную функциональную запись, мы получаем хорошую производительность.

Существуют и другие разновидности объектов функций. Например, некоторые объекты функций позволяют вызвать определенную функцию класса для каждого элемента коллекции:

```
for_each (coll.begin(), coll.end(),           // Интервал
          mem_fun_ref(&Person::save));        // Операция
```

Объект функции *mem_fun_ref* вызывает заданную функцию класса для того элемента, для которого этот объект вызывается. В приведенном примере для каждого элемента коллекции *coll* вызывается функция *save()* класса *Person*. Разумеется, эта конструкция работает только в том случае, если элемент относится к типу *Person* или производному от него.

На с. 305 перечислены и более подробно описаны все стандартные объекты функций, функциональные адаптеры и аспекты функциональной композиции. Кроме того, в этом разделе рассказано, как написать собственный объект функции.

Элементы контейнеров

Элементы контейнеров должны удовлетворять ряду требований, потому что контейнеры работают с ними по определенным правилам. Настоящий раздел посвящен требованиям, предъявляемым к элементам. Кроме того, мы разберемся, к каким последствиям приводит создание контейнером внутренних копий элементов.

Требования к элементам контейнеров

Контейнеры, итераторы и алгоритмы STL оформлены в виде шаблонов. Это означает, что они могут работать с любыми типами, как заранее определенными, так и пользовательскими. Впрочем, специфика выполняемых операций накладывает ряд требований на элементы контейнеров STL. Таких требований три.

- *Требуется возможность копирования* элемента копирующим конструктором. Созданная копия должна быть эквивалентна оригиналу. Это означает, что любая проверка на равенство должна считать копию и оригинал равными, а поведение копии не должно отличаться от поведения оригинала.

Все контейнеры создают внутренние копии своих элементов и возвращают их временные копии. Копирующий конструктор вызывается очень часто, поэтому он должен обладать хорошим быстродействием (это не требование, а всего лишь условие, повышающее эффективность работы контейнера). Если копирование объектов занимает слишком много времени, попробуйте пре-

дотвратить копирование, используя контейнер со ссылочной семантикой. За подробностями обращайтесь на с. 226.

- *Требуется возможность присваивания* элемента оператором присваивания. Контейнеры и алгоритмы используют оператор присваивания для замены старых элементов новыми.
- *Требуется возможность уничтожения* элемента деструктором. Контейнеры уничтожают свои внутренние копии элементов при удалении этих элементов из контейнера. Следовательно, деструктор не должен быть закрытым (*private*). Кроме того, как обычно в C++, деструктор не должен инициировать исключения, иначе возможны абсолютно непредсказуемые последствия.

Эти три операции — копирование, присваивание, уничтожение — автоматически генерируются для любого класса. Следовательно, любой класс автоматически удовлетворяет требованиям, если для него не были определены специальные версии этих операций или их нормальная работа не нарушается другими членами класса.

Кроме того, к элементам могут предъявляться дополнительные требования¹.

- Для некоторых функций классов последовательных контейнеров *требуется конструктор по умолчанию*. Например, можно создать непустой контейнер или увеличить количество элементов без указания значений новых элементов. Такие элементы создаются вызовом конструктора по умолчанию для соответствующего типа.
- Для некоторых операций *требуется определить проверку на равенство* оператором `==`. Такая необходимость особенно часто возникает при поиске.
- Для ассоциативных контейнеров *требуется, чтобы элементы поддерживали критерий сортировки*. По умолчанию используется оператор `<`, вызываемый объектом функции `less<>`.

Семантика значений и ссылочная семантика

Все контейнеры создают внутренние копии своих элементов и возвращают эти копии. Следовательно, элементы контейнера равны, но не идентичны объектам, заносимым в контейнер. При модификации объектов, являющихся элементами контейнера, вы модифицируете копию, а не исходный объект.

Копирование значений означает, что контейнеры STL поддерживают *семантику значений*. Они содержат значения вставляемых объектов, а не сами объекты. Но на практике также возникает необходимость в *ссылочной семантике*, при которой контейнеры содержат ссылки на объекты, являющиеся их элементами.

¹ В некоторых старых системах C++ приходится обеспечивать выполнение этих дополнительных требований, даже если они не используются в программе. Например, некоторые реализации `vector` требуют обязательного наличия деструктора по умолчанию для элементов. В других реализациях всегда должны присутствовать оператор сравнения. В соответствии со стандартом эти требования незаконны; скорее всего, в будущем подобные ограничения будут устраниены.

Принятый в STL подход, то есть поддержка только семантики значений, имеет свои положительные и отрицательные стороны. К достоинствам такого подхода относятся:

- простота копирования элементов;
- при использовании ссылок часто возникают ошибки (программист должен следить за тем, чтобы ссылки не относились к несуществующим объектам, кроме того, необходимо предусмотреть обработку возможных циклических ссылок).

Недостатки:

- при отказе от ссылочной семантики копирование элементов может выполняться неэффективно или становится невозможным;
- невозможность одновременного присутствия объектов в нескольких контейнерах.

На практике нужны оба подхода: копии, независимые от исходных данных (семантика значений), и копии, ссылающиеся на исходные данные и изменяющиеся вместе с ними (ссылочная семантика). К сожалению, стандартная библиотека C++ не поддерживает ссылочную семантику. Впрочем, ее можно реализовать в контексте семантики значений.

Наиболее очевидная реализация ссылочной семантики основана на использовании указателей как элементов контейнеров¹. Тем не менее обычным указателям присущи хорошо известные недостатки. Например, объект, на который ссылается указатель, оказывается несуществующим, или операция сравнения работает не так, как предполагалось, потому что вместо объектов сравниваются указатели на них. Следовательно, при использовании обычных указателей в элементах контейнеров нужно действовать очень осторожно.

Более разумное решение основано на применении *умных указателей* — объектов, поддерживающих интерфейс указателей, но выполняющих дополнительную внутреннюю проверку или другие операции. Но здесь возникает важный вопрос: насколько умными должны быть эти указатели? В стандартную библиотеку C++ входит класс умного указателя `auto_ptr` (см. с. 54), который на первый взгляд мог бы пригодиться. К сожалению, этот класс не подходит, поскольку не удовлетворяет одному из основных требований к элементам контейнеров, а именно: после копирования или присваивания объектов класса `auto_ptr` оригинал и копия не эквивалентны. Исходный объект `auto_ptr` изменяется, потому что значение *передается*, а не копируется (см. с. 59 и 62). На практике это означает, что сортировка и даже простой вывод элементов контейнера может привести к их уничтожению. Следовательно, объекты `auto_ptr` не должны использоваться как элементы контейнеров (в системе C++, соответствующей стандарту, такие попытки приводят к ошибкам компиляции). Дополнительная информация приведена на с. 59.

Чтобы реализовать ссылочную семантику для контейнеров STL, вам придется написать собственный класс умного указателя. Но будьте внимательны: используйте

¹ Реализация ссылочной семантики с применением указателей хорошо знакома программистам С. В языке С аргументы функций передаются только по значению, поэтому для передачи по ссылке приходится использовать указатели.

зование умного указателя с подсчетом ссылок (умного указателя, который автоматически уничтожает связанный объект при уничтожении последней ссылки на него) вызовет немало проблем. Например, при прямом доступе к элементам возможна модификация их значений, пока они находятся в контейнерах. В ассоциативном контейнере это приведет к нарушению порядка следования элементов, а это недопустимо.

На с. 226 приведена дополнительная информация о контейнерах со ссылочной семантикой и продемонстрирован один из возможных способов реализации ссылочной семантики для контейнеров STL на базе умных указателей с подсчетом ссылок.

Ошибки и исключения внутри STL

Ошибки случаются, хотим мы того или нет. Они могут быть обусловлены как действиями программы (а точнее, программиста), так и контекстом или рабочей средой программы (например, нехваткой памяти). Обе разновидности ошибок обрабатываются при помощи исключений (начальные сведения об исключениях приводятся на с. 31). Данный раздел посвящен принципам обработки ошибок и исключений в STL.

Обработка ошибок

При проектировании STL главным приоритетом была максимальная производительность, а не безопасность. Проверка ошибок требует времени, поэтому в STL она практически не выполняется. Если вы умсете программировать без ошибок, все замечательно, а если нет — дело кончится катастрофой. Перед включением библиотеки STL в стандартную библиотеку C++ обсуждался вопрос о том, не нужно ли расширить обработку ошибок. Большинство высказались против по двум причинам.

- Обработка ошибок снижает быстродействие, а высокая скорость работы попрежнему остается основной целью большинства программ. Как упоминалось выше, быстродействие было одним из приоритетов при проектировании STL.
- Тот, кто ставит на первое место надежность, может добиться своего при помощи интерфейсных оболочек или специальных версий STL. С другой стороны, невозможно повысить быстродействие за счет отказа от проверки ошибок, если эта проверка включена во все базовые операции. Например, если при любой выборке элемента по индексу проверяется, принадлежит ли индекс к интервалу допустимых значений, вам не удастся написать собственную процедуру индексации без проверки. А вот обратная ситуация вполне возможна.

В результате проверка ошибок в STL возможна, но не обязательна.

В спецификации стандартной библиотеки C++ указано, что любое использование STL, нарушающее предварительные условия, приводит к непредсказуемому поведению. Следовательно, при недействительном индексе, итераторе или

интервале может произойти все что угодно. Если не использовать безопасную версию STL, обычно дело кончается нарушением защиты памяти с неприятными побочными эффектами и даже сбоем программы. В некотором смысле применение библиотеки STL так же чревато ошибками, как применение указателей в языке С. Найти такие ошибки иногда бывает очень трудно, особенно если нет безопасной версии STL.

В частности, для нормальной работы STL должны выполняться перечисленные ниже условия.

- Действительность итераторов. Например, перед использованием итераторы должны инициализироваться. Следует помнить, что итераторы могут стать недействительными вследствие побочных эффектов других операций. В частности, в векторах и деках это может произойти при вставке, удалении или перемещении элементов.
- Конечный итератор не связан с элементом контейнера, поэтому вызов операторов * и -> для него недопустим. В частности, это относится к возвращаемым значениям функций end() и rend() контейнерных классов.
- Действительность интервалов:
 - итераторы, определяющие интервал, должны относиться к одному контейнеру;
 - второй итератор должен быть достижим в результате перебора элементов, начиная от первого.
- При использовании нескольких интервалов-источников второй и последующие интервалы должны содержать не меньше элементов, чем первый интервал.
- Количество элементов в приемном интервале должно быть достаточным для перезаписи; в противном случае необходимо использовать итераторы вставки.

Некоторые распространенные ошибки продемонстрированы в следующем примере:

```
// stl/iterbug1.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> coll1;      // Пустая коллекция
    vector<int> coll2;      // Пустая коллекция

    /* ОШИБКА:
     * - начало находится за концом интервала
     */
    vector<int>::iterator pos = coll1.begin();
    reverse (++pos, coll1.end());

    // Вставка элементов со значениями от 1 до 9 в coll2
    for (int i=1; i<=9; ++i) {
```

```

    coll2.push_back (i);

}

/* ОШИБКА:
 * - перезапись несуществующих элементов
 */
copy (coll2.begin(), coll2.end(),      // Источник
      coll1.begin());                  // Приемник

/* ОШИБКА:
 * - перепутаны коллекции
 * - перепутаны begin() и end()
 */
copy (coll1.begin(), coll2.end(),      // Источник
      coll1.end());                  // Приемник
}

```

Все указанные ошибки происходят не на стадии компиляции, а во время выполнения программы, поэтому они приводят к непредсказуемым последствиям.

Библиотека STL предоставляет массу возможностей для ошибок. Она вовсе не обязана защищать вас от самого себя, поэтому желательно (по крайней мере, на время разработки программы) использовать безопасную версию STL. Первая безопасная версия STL была разработана Кеем Хорстманом (Cay Horstmann)¹. К сожалению, многие поставщики предоставляют версию STL, основанную на исходном варианте кода, в котором обработка ошибок не предусмотрена. Впрочем, ситуация постепенно улучшается. Современная безопасная версия STL распространяется бесплатно практически для всех платформ (<http://www.stlport.org/>).

Обработка исключений

Проверка логических ошибок в STL практически отсутствует, поэтому сама библиотека STL почти не генерирует исключения, связанные с логикой. Фактически существует только одна функция, для которой в стандарте прямо указано на возможность возникновения исключения: речь идет о функции `at()` векторов и деков (проверяемой версии оператора индексирования). Во всех остальных случаях стандарт требует лишь стандартных исключений типа `bad_alloc` при нехватке памяти или исключений, возникающих при пользовательских операциях.

Когда генерируются исключения и что при этом происходит с компонентами STL? В течение долгого времени, пока шел процесс стандартизации, строгих правил на этот счет не существовало. В сущности, любое исключение приводило к непредсказуемым последствиям. Даже уничтожение контейнера STL после того, как во время выполнения одной из поддерживаемых им операций произошло исключение, тоже приводило к непредсказуемым последствиям (например, аварийному завершению программы). Из-за этого библиотека STL оказывалась бесполезной там, где требовалось гарантированное четко определенное поведение, потому что не была предусмотрена даже возможность раскрутки стека.

¹ За безопасной версией STL Кея Хорстмана обращайтесь по адресу <http://www.horstmann.com/safestl.html>.

Обработка исключений стала одним из последних вопросов, обсуждавшихся в процессе стандартизации. Найти хорошее решение оказалось нелегко. На это понадобилось много времени, что объяснялось двумя основными причинами.

- Было очень трудно определить, какую степень безопасности должна обеспечивать стандартная библиотека C++. Напрашивается мысль, что всегда следует обеспечивать максимальный уровень из всех возможных. Например, можно потребовать, чтобы вставка нового элемента в произвольной позиции вектора либо завершалась успешно, либо не вносила изменений. Однако исключение может произойти в процессе копирования элементов, расположенных после позиции вставки, на следующую позицию для освобождения места под новый элемент; в этом случае полное восстановление невозможно. Чтобы добиться указанной цели, вставку пришлось бы реализовать с копированием *всех* элементов вектора в новую область памяти, что серьезно отразилось бы на быстродействии. Если приоритетной задачей проектирования является высокое быстродействие (как в случае STL), обеспечить идеальную обработку исключений для всех возможных ситуаций все равно не удастся. Приходится искать компромисс между безопасностью и скоростью работы.
- Многие беспокоились о том, что присутствие кода обработки исключений отрицательно скажется на быстродействии. Это противоречило бы главной цели проектирования — обеспечению максимального быстродействия. Впрочем, разработчики компиляторов утверждают, что в принципе обработка исключений реализуется без сколько-нибудь заметного снижения быстродействия (причем такие реализации уже существуют). Несомненно, лучше иметь гарантированные четко определенные правила обработки исключений без заметного снижения быстродействия, чем рисковать системными сбоями из-за исключений.

В результате этих обсуждений стандартная библиотека C++ теперь предоставляет *базовую гарантию* безопасности исключений¹: возникновение исключений в стандартной библиотеке C++ не приводит к утечке ресурсов или нарушению контейнерных инвариантов.

К сожалению, во многих случаях этого недостаточно. Довольно часто требуется более твердая гарантия того, что при возникновении исключения произойдет возврат к состоянию, имевшему место перед началом операции. О таких операциях говорят как об *атомарных* по отношению к исключениям, а если воспользоваться терминологией баз данных, можно сказать, что эти операции должны обеспечивать возможность либо *приятия*, либо *отката*, то есть *транзакционную безопасность*.

В том, что касается этих требований, стандартная библиотека C++ теперь гарантирует две вещи.

- Для *всех узловых контейнеров* (списки, множества, мультимножества, отображения и мультиотображения) неудачная попытка создания узла просто оставляет контейнер в прежнем состоянии. Более того, удаление узла не может завершиться неудачей (если исключение не произойдет в деструкторе). Однако при вставке нескольких элементов в ассоциативный контейнер из-за необходимости сохранения порядка сортировки обеспечивать полное восстановление

¹ Спасибо Дэйву Абрахамсу (Dave Abrahams) и Грэгу Колвибу (Greg Colvin) за их работу в области безопасности исключений в стандарте C++, а также за информацию, полученную от них по этой теме.

было бы непрактично. Таким образом, только одиночная вставка в ассоциативные контейнеры обеспечивает транзакционную безопасность (то есть либо завершается успешно, либо не вносит изменений). Кроме того, гарантируется, что все операции удаления (как одного, так и нескольких элементов) всегда завершаются успешно.

Для списков даже вставка нескольких элементов обладает транзакционной безопасностью. Более того, любые операции со списками, кроме `remove()`, `remove_if()`, `merge()`, `sort()` и `unique()`, либо завершаются успешно, либо не вносят изменений. Для некоторых из перечисленных операций стандартная библиотека C++ предоставляет условные гарантии (см. с. 180). Отсюда можно сделать вывод: если вам требуется контейнер, обладающий транзакционной безопасностью, выбирайте список.

- **Контейнеры на базе массивов** (векторы и деки) не обеспечивают полного восстановления при вставке элементов. Для этого пришлось бы копировать все элементы, находящиеся за позицией вставки, а на обеспечение полного восстановления для всех операций копирования ушло бы слишком много времени. Тем не менее операции присоединения и удаления элементов в конце контейнера не требуют копирования существующих элементов, поэтому при возникновении исключений гарантируется откат (возвращение к прежнему состоянию). Более того, если элементы относятся к типу, у которого операции копирования (копирующий конструктор и оператор присваивания) не генерируют исключений, то любые операции с этими элементами либо завершаются успешно, либо не вносят изменений.

Более подробный перечень всех контейнерных операций, обеспечивающих в отношении исключений более твердые гарантии, приведен на с. 254.

Учите, что все гарантии основаны на запрете исключений в деструкторах (который в C++ должен выполняться всегда). Стандартная библиотека C++ соблюдает это требование; его должны соблюдать и прикладные программисты.

Если вам понадобится контейнер с полными гарантиями транзакционной безопасности, используйте либо список (без вызова функций `sort()` и `unique()`), либо ассоциативный контейнер (без вызова многоэлементных операций вставки). Тогда вам не придется создавать копии данных перед модификацией, чтобы предотвратить возможную потерю данных. Копирование контейнеров иногда обходится очень дорого.

Если вы не используете узловой контейнер, но нуждаетесь в полноценной поддержке транзакционной безопасности, для всех критических операций придется делать промежуточные копии данных. Например, следующая функция обеспечивает почти безопасную вставку значения в заданную позицию произвольного контейнера:

«Почти» означает, что эта функция не идеальна. Дело в том, что функция `swap()` тоже может генерировать исключение, если оно имсется место в критерии сравнения ассоциативного контейнера. Как видите, безупречная обработка исключений — дело весьма непростое.

Расширение STL

Библиотека STL проектировалась с расчетом на возможность расширения практически в любом направлении. Программист может создавать и использовать собственные контейнеры, итераторы, алгоритмы и объекты функций, удовлетворяющие определенным требованиям. Кроме того, в стандартной библиотеке C++ не поддерживаются некоторые полезные возможности, поскольку в какой-то момент комитет по стандартизации был вынужден прекратить прием новых предложений и сосредоточиться на приведении в порядок того, что есть; иначе работа продолжалась бы до бесконечности.

Самое заметное упущение в STL — отсутствие такого типа контейнера, как хэш-таблица. Просто предложение о включении хэш-таблиц в стандартную библиотеку C++ поступило слишком поздно. Тем не менее весьма вероятно, что новые версии стандарта будут содержать те или иные формы хэшей. Большинство реализаций библиотеки C++ уже содержат хэш-контейнеры, но, к сожалению, все они реализованы по-разному. За дополнительной информацией обращайтесь на с. 225.

Среди других полезных расширений стоит отметить дополнительные объекты функций (см. с. 313), итераторы (см. с. 291), контейнеры (см. с. 221) и алгоритмы (см. с. 289).

6 Контейнеры STL

В этой главе, которая продолжает тему, начатую в главе 5, приведены подробные описания контейнеров STL. Начинается она с обзора общих возможностей и операций всех контейнерных классов, после чего каждый контейнерный класс рассматривается в отдельности. В каждом случае представлены внутренние структуры данных, основные операции контейнера и их эффективность. Также показано, как использовать различные операции контейнера; нетривиальные случаи поясняются примерами. Кроме того, типичными примерами использования контейнеров завершаются все разделы, посвященные конкретным контейнерам. Далее обсуждается интересный вопрос: в каких ситуациях следует применять тот или иной контейнер? Сравнительный анализ общих возможностей, достоинств и недостатков разных типов контейнеров поможет подобрать оптимальный контейнер для конкретной ситуации. Глава завершается подробным описанием всех членов контейнерных классов. Это описание было задумано как своего рода справочное руководство. В нем перечислены все нюансы интерфейса контейнера и приводятся точные сигнатуры всех контейнерных операций. Там, где это уместно, приводятся перекрестные ссылки на похожие или вспомогательные алгоритмы.

Стандартная библиотека C++ также содержит ряд специализированных контейнерных классов — это так называемые *контейнерные адаптеры* (стек, очередь, приоритетная очередь), *битовые поля* и *массивы значений*. Все эти контейнеры обладают специальным интерфейсом, не соответствующим общим требованиям к контейнерам STL, поэтому они рассматриваются отдельно¹. Контейнерные адаптеры и битовые поля рассматриваются в главе 10. Массивы значений описаны на с. 525.

Общие возможности и операции

Общие возможности контейнеров

Здесь сформулированы общие возможности всех контейнерных классов STL. В большинстве случаев речь идет о *требованиях*, которые должны выполняться всеми контейнерами STL. Ниже перечислены три основных требования.

¹ Традиционно контейнерные адаптеры считаются частью STL. Тем не менее на концептуальном уровне адаптеры не входят в иерархию STL, а «всего лишь» используют ее.

- Контейнеры должны поддерживать семантику значений вместо ссылочной семантики. При вставке элемента контейнер создает его внутреннюю копию, вместо того чтобы сохранять ссылку на внешний объект. Следовательно, элементы контейнера STL должны поддерживать копирование. Если объект, который требуется сохранить в контейнере, не имеет открытого копирующего конструктора или копирование объекта нежелательно (например, если оно занимает слишком много времени или элементы должны принадлежать сразу нескольким контейнерам), в контейнер заносится указатель или объект указателя, ссылающийся на этот объект. Проблема подробно рассматривается на с. 144.
- Элементы в контейнере располагаются в определенном порядке. Это означает, что при повторном переборе с применением итератора порядок перебора элементов должен оставаться прежним. В каждом типе контейнера определены операции, возвращающие итераторы для перебора элементов. Итераторы представляют собой основной интерфейс для работы алгоритмов STL.
- В общем случае операции с элементами контейнеров небезопасны. Вызывающая сторона должна проследить за тем, чтобы параметры операции соответствовали требованиям. Нарушение правил (например, использование недействительного индекса) приводит к непредсказуемым последствиям. Обычно STL не генерирует исключений в своем коде. Но если исключение генерируется пользовательскими операциями, вызываемыми контейнером STL, ситуация меняется. За подробностями обращайтесь на с. 148.

Общие операции над контейнерами

В табл. 6.1 перечислены операции, общие для всех контейнеров. Общие операции подчиняются правилам, упомянутым в предыдущем подразделе. Некоторые из них более подробно описаны далее.

Таблица 6.1. Общие операции контейнерных классов

Операция	Описание
<code>ContType c</code>	Создает пустой контейнер, не содержащий элементов
<code>ContType c1(c2)</code>	Создает копию контейнера того же типа
<code>ContType c(beg,end)</code>	Создает контейнер и инициализирует его копиями всех элементов в интервале [beg,end)
<code>c.\sim ContType ()</code>	Удаляет все элементы и освобождает память
<code>c.size()</code>	Возвращает фактическое количество элементов
<code>c.empty()</code>	Проверяет, пуст ли контейнер (эквивалент <code>size() == 0</code> , но иногда выполняется быстрее)
<code>c.max_size()</code>	Возвращает максимально возможное количество элементов
<code>c1 == c2</code>	Проверяет равенство <code>c1</code> и <code>c2</code>
<code>c1 != c2</code>	Проверяет неравенство <code>c1</code> и <code>c2</code> (эквивалентно <code>!(c1 == c2)</code>)

Таблица 6.1 (продолжение)

Операция	Описание
<code>c1 < c2</code>	Проверяет, что <code>c1</code> меньше <code>c2</code>
<code>c1 > c2</code>	Проверяет, что <code>c1</code> больше <code>c2</code> (эквивалент <code>c2 < c1</code>)
<code>c1 <= c2</code>	Проверяет, что <code>c1</code> не больше <code>c2</code> (эквивалент <code>!(c2 < c1)</code>)
<code>c1 >= c2</code>	Проверяет, что <code>c1</code> не меньше <code>c2</code> (эквивалент <code>!(c1 < c2)</code>)
<code>c1 = c2</code>	Присваивает <code>c1</code> все элементы <code>c2</code>
<code>c1.swap(c2)</code>	Меняет местами содержимое <code>c1</code> и <code>c2</code>
<code>swap(c1,c2)</code>	То же, но в форме глобальной функции
<code>c.begin()</code>	Возвращает итератор для первого элемента
<code>c.end()</code>	Возвращает итератор для позиции за последним элементом
<code>c.rbegin()</code>	Возвращает обратный итератор для первого элемента при переборе в обратном направлении
<code>c.rend()</code>	Возвращает обратный итератор для позиции за последним элементом при переборе в обратном направлении
<code>c.insert(pos,elem)</code>	Вставляет копию <code>elem</code> (с разными вариантами возвращаемого значения и интерпретацией первого аргумента)
<code>c.erase(beg,end)</code>	Удаляет все элементы из интервала <code>[beg,end]</code> (некоторые контейнеры возвращают следующий элемент после удаленного интервала)
<code>c.clear()</code>	Удаляет из контейнера все элементы (контейнер остается пустым)
<code>c.get_allocator()</code>	Возвращает модель памяти контейнера

Инициализация

Любой контейнерный класс содержит конструктор по умолчанию, копирующий конструктор и деструктор. Кроме того, предусмотрена возможность инициализации контейнера элементами из заданного интервала. Соответствующий конструктор позволяет инициализировать контейнер элементами другого контейнера, массива или просто элементами, прочитанными из стандартного входного потока данных. Такие конструкторы оформляются в виде шаблонных функций класса (см. с. 28), поэтому различия возможны не только в типе контейнера, но и в типе элементов (при условии автоматического преобразования исходного типа элемента к итоговому типу)¹. Несколько примеров инициализации:

○ Инициализация элементами другого контейнера:

```
std::list<int> l;           // l - связанный список int
...
// Копирование всех элементов списка в вектор с приведением к типу float
std::vector<float> c(l.begin(),l.end());
```

¹ Если система не поддерживает шаблонные функции классов, обычно разрешается инициализация только при полном соответствии типов. В таких случаях можно воспользоваться алгоритмом `copy()`. Пример приведен на с. 196.

- Инициализация элементами массива:

```
int array[] = { 2, 3, 17, 33, 45, 77 }:  
...  
// Копирование всех элементов массива в множество  
std::set<int> c(array, array+sizeof(array)/sizeof(array[0]));
```

- Инициализация с применением стандартного входного потока данных:

```
// Чтение элементов типа int в дек из стандартного ввода  
std::deque<int> c((std::istream_iterator<int>(std::cin)),  
                   (std::istream_iterator<int>()));
```

Не забудьте заключить аргументы инициализатора в дополнительные круглые скобки. Без них выражение работает совсем не так, как предполагалось. Скорее всего, все кончается странными предупреждениями или ошибками в следующих командах. Рассмотрим ту же команду без дополнительных круглых скобок:

```
std::deque<int> c(std::istream_iterator<int>(std::cin),  
                   std::istream_iterator<int>());
```

В этом варианте `c` объявляет *функцию*, возвращающую тип `deque<int>`. Ее первый параметр `cin` относится к типу `istream_iterator<int>`, а второй безымянный параметр — к типу «функция, вызываемая без аргументов и возвращающая `istream_iterator<int>`». Эта конструкция синтаксически действительна и как объявление, и как выражение. Следовательно, в соответствии с правилами языка она интерпретируется как объявление. При введении дополнительных круглых скобок инициализатор перестает соответствовать синтаксису объявления¹.

В принципе эта методика также применяется при присваивании или вставке элементов из другого интервала. Однако в этих операциях интерфейс либо отличается наличием дополнительных аргументов, либо поддерживается не всеми контейнерными классами.

Операции проверки размера

Все контейнерные классы поддерживают три операции, связанные с проверкой размера.

- `size()`. Функция возвращает текущее количество элементов в контейнере.
- `empty()`. Сокращенная форма проверки нулевого текущего количества элементов в контейнерах (`size() == 0`). Однако функция `empty()` может быть реализована более эффективно, поэтому по возможности следует использовать именно ее.
- `max_size()`. Функция возвращает максимальное количество элементов, которые могут содержаться в контейнере. Значение зависит от реализации. Например, все элементы вектора обычно хранятся в одном блоке памяти, что может обусловить дополнительные ограничения на РС. В общем случае `max_size()` совпадает с максимальным значением типа индекса.

¹ Спасибо Джону Спайсеру (John H. Spicer) из EDG за это объяснение.

Сравнения

Обычные операторы сравнения `==`, `!=`, `<`, `<=`, `>` и `>=` определяются по следующим трем правилам.

- Оба контейнера должны относиться к одному типу.
- Два контейнера равны, если их элементы совпадают и следуют в одинаковом порядке. Проверка на равенство элементов выполняется оператором `==`.
- Отношение «меньше/больше» между контейнерами проверяется по лексикографическому критерию (см. с. 356).

Для сравнения разнотипных контейнеров применяются алгоритмы сравнения, описанные на с. 352.

Присваивание и функция swap

В процессе присваивания контейнеров все элементы контейнера-источника копируются, а все элементы контейнера-приемника удаляются. Таким образом, присваивание контейнеров является относительно дорогостоящей операцией.

Если контейнеры относятся к одному типу, а источник в дальнейшем не требуется, существует простой способ оптимизации: воспользуйтесь функцией `swap()`. Функция `swap()` работает гораздо эффективнее, потому что использует только внутренние данные контейнеров. Более того, она переставляет только внутренние указатели со ссылками на данные (элементы, распределитель памяти и критерий сортировки, если он есть). Следовательно, функция `swap()` при присваивании заведомо выполняется с постоянной, а не с линейной сложностью.

Векторы

Вектором называется абстрактная модель, имитирующая динамический массив при операциях с элементами (рис. 6.1). Однако стандарт не утверждает, что в реализации вектора должен использоваться именно динамический массив. Скорее этот выбор обусловлен ограничениями и требованиями к сложности операций.

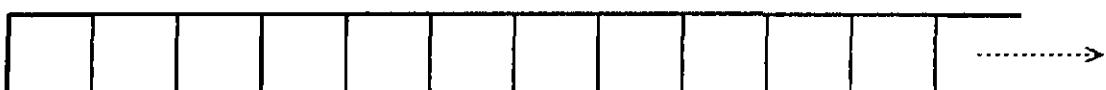


Рис. 6.1. Структура вектора

Чтобы использовать вектор в программе, необходимо включить в нее заголовочный файл `<vector>`¹:

```
#include <vector>
```

¹ В исходной версии STL вектор определялся в заголовочном файле `<vector.h>`.

Тип вектора определяется как шаблон класса в пространстве имён std:

```
namespace std {
    template <class T,
              class Allocator = allocator<T> >
    class vector;
}
```

Элементы вектора относятся к произвольному типу T, поддерживающему присваивание и копирование. Необязательный второй параметр шаблона определяет модель памяти (см. главу 15). По умолчанию используется модель allocator, определенная в стандартной библиотеке C++¹.

Возможности векторов

Элементы вектора копируются во внутренний динамический массив. Элементы всегда хранятся в определенном порядке; следовательно, вектор относится к категории *упорядоченных коллекций*. Вектор обеспечивает *произвольный доступ* к своим элементам. Это означает, что обращение к любому элементу с известной позицией выполняется напрямую и с постоянным временем. Итераторы векторов являются итераторами произвольного доступа, что позволяет применять к векторам все алгоритмы STL.

Операции присоединения и удаления элементов в конце вектора выполняются с высоким быстродействием. Если элементы вставляются или удаляются в середине или в начале, быстродействие снижается, поскольку все элементы в последующих позициях приходится перемещать на новое место. На самом деле для каждого последующего элемента вызывается оператор присваивания.

Один из способов повышения быстродействия векторов заключается в выделении для вектора большего объема памяти, чем необходимо для хранения всех элементов. Чтобы эффективно и правильно использовать векторы, нужно понимать, как связаны между собой размер и емкость вектора.

Векторы поддерживают стандартные операции проверки размера size(), empty() и max_size() (см. с. 155). К ним добавляется функция capacity(), возвращающая максимальное количество элементов, которые могут храниться в текущей выделенной памяти. Если количество элементов превысит значение, возвращаемое функцией capacity(), вектору придется перераспределить свою внутреннюю память.

Емкость вектора необходимо учитывать по двум причинам:

- в результате перераспределения памяти становятся недействительными все ссылки, указатели и итераторы элементов вектора;
- на перераспределение памяти требуется время.

Следовательно, если программа поддерживает указатели, ссылки или итераторы для вектора, а также при важности быстродействия нужно помнить о емкости вектора.

¹ В системах, не поддерживающих значения по умолчанию для параметров шаблонов, второй параметр обычно отсутствует.

Чтобы предотвратить перераспределение памяти, можно заранее зарезервировать некоторую емкость функцией `reserve()`. Тем самым гарантируется, что ссылки останутся действительными, пока зарезервированная емкость не будет исчерпана:

```
std::vector<int> v;      // Создание пустого вектора
v.reserve(80);           // Резервирование памяти для 80 элементов
```

В другом варианте вектор инициализируется достаточным количеством элементов, для чего конструктору передаются дополнительные аргументы. Например, если передать числовое значение, оно интерпретируется как начальный размер вектора:

```
std::vector<T> v(5);    // Создание вектора и его инициализация
                        // пятью значениями (с пятикратным вызовом
                        // конструктора по умолчанию типа T)
```

Конечно, для этого при определении типа элементов должен быть определен конструктор по умолчанию. Но учтите, что для сложных типов даже при наличии конструктора по умолчанию инициализация требует времени. Если элементы инициализируются только для того, чтобы зарезервировать память, лучше воспользоваться функцией `reserve()`.

Концепция емкости векторов сходна с аналогичной концепцией для строк (см. с. 468), но с одним существенным различием: в отличие от строк функция `reserve()` не может вызываться для уменьшения емкости векторов. Вызов `reserve()` с аргументом, меньшим текущей емкости вектора, игнорируется. Более того, способы оптимизации по скорости и затратам памяти определяются реализацией. Это означает, что реализация может увеличивать емкость с большими прращениями. Для предотвращения внутренней фрагментации многие реализации выделяют полный блок памяти (около 2 Кбайт) при первой вставке, если ранее не вызывалась функция `reserve()`. Если программа работает с множеством векторов, содержащих малое количество элементов, такая стратегия приводит к неэффективному расходованию памяти.

Емкость вектора напрямую никогда не уменьшается, поэтому ссылки, указатели и итераторы заведомо остаются действительными даже при удалении элементов (при условии, что они ссылкаются на позицию, расположенную перед модифицированными элементами). Однако вставка может привести к тому, что ссылки, указатели и итераторы станут недействительными.

Впрочем, способ косвенного уменьшения емкости вектора все же существует: если поменять местами содержимое двух векторов функцией `swap()`, при этом также поменяются их емкости. Следующая функция уменьшает емкость вектора с сохранением элементов:

```
template <class T>
void shrinkCapacity(std::vector<T>& v)
{
    std::vector<T> tmp(v); // Копирование элементов в новый вектор
```

```
v.swap(tmp);           // Перестановка внутренних данных векторов
}
```

Емкость вектора можно уменьшить даже без вызова этой функции, просто выполнив следующую команду¹:

```
// Уменьшение емкости вектора v для типа T
std::vector<T>(v).swap(v);
```

Но не следует забывать, что после вызова функции `swap()` все ссылки, указатели и итераторы переключаются на новый контейнер. Они продолжают ссылаться на те элементы, на которые они ссылались первоначально. Значит, после вызова функции `shrinkCapacity()` все ссылки, указатели и итераторы становятся недействительными.

Операции над векторами

Операции создания, копирования и уничтожения

В табл. 6.2 перечислены конструкторы и деструктор векторов. Векторы создаются с инициализацией элементов или без нее. Если передается только размер, элементы создаются конструктором по умолчанию. Обратите внимание: явный вызов конструктора по умолчанию также инициализирует базовые типы (в частности, `int`) нулями; эта особенность языка описана на с. 30. Некоторые из возможных источников инициализации упоминаются на с. 154.

Таблица 6.2. Конструкторы и деструкторы векторов

Операция	Описание
<code>vector<Elem> c</code>	Создает пустой вектор, не содержащий ни одного элемента
<code>vector<Elem> c1(c2)</code>	Создает копию другого вектора того же типа (с копированием всех элементов)
<code>vector<Elem> c(n)</code>	Создает вектор из n элементов, создаваемых конструктором по умолчанию
<code>vector<Elem> c(n, elem)</code>	Создает вектор, инициализируемый n копиями элемента elem
<code>vector<Elem> c(beg, end)</code>	Создает вектор, инициализируемый элементами интервала [beg,end)
<code>c~vector<Elem>()</code>	Уничтожает все элементы и освобождает память

Немодифицирующие операции над векторами

В табл. 6.3 перечислены все операции, выполняемые без модификации вектора. Дополнительные замечания приведены на с. 154 и 157.

¹ Компилятор может посчитать эту команду неправильной, потому что в ней неконстантная функция класса вызывается для временного значения. Однако на самом деле стандарт C++ позволяет вызывать неконстантные функции классов для временных значений.

Таблица 6.3. Немодифицирующие операции над векторами

Операция	Описание
c.size()	Возвращает фактическое количество элементов
c.empty()	Проверяет, пуст ли контейнер (эквивалент size()==0, но иногда выполняется быстрее)
c.max_size()	Возвращает максимально возможное количество элементов
capacity()	Возвращает максимально возможное количество элементов без перераспределения памяти
reserve()	Увеличивает емкость вектора, если текущая емкость меньше заданной ¹
c1 == c2	Проверяет равенство c1 и c2
c1 != c2	Проверяет неравенство c1 и c2 (эквивалент !(c1==c2))
c1 < c2	Проверяет, что c1 меньше c2
c1 > c2	Проверяет, что c1 больше c2 (эквивалент c2<c1)
c1 <= c2	Проверяет, что c1 не больше c2 (эквивалент !(c2<c1))
c1 >= c2	Проверяет, что c1 не меньше c2 (эквивалент !(c1<c2))

Присваивание

В табл. 6.4 перечислены операции присваивания новых элементов с одновременным удалением старых. Набор функций `assign()` соответствует набору конструкторов класса. При присваивании могут использоваться разные источники (контейнеры, массивы, стандартный входной поток данных) — по аналогии с источниками, используемыми при вызове конструкторов (см. с. 154).

Таблица 6.4. Операции присваивания для векторов

Операция	Описание
c1 = c2	Присваивает c1 все элементы c2
c.assign(n, elem)	Присваивает n копий заданного элемента
c.assign(beg, end)	Присваивает элементы интервала [beg, end)
c1.swap(c2)	Меняет местами содержимое c1 и c2
swap(c1, c2)	То же, но в форме глобальной функции

Все операции присваивания вызывают конструктор по умолчанию, копирующий конструктор, оператор присваивания и/или деструктор типа элемента в зависимости от того, как изменяется количество элементов в контейнере. Пример:

```
std::list<Elem> l;
std::vector<Elem> coll;
```

¹ Функция `reserve()` изменяет вектор, поскольку в результате ее выполнения становятся недействительными ссылки, указатели и итераторы. Тем не менее она включена в таблицу, потому что вызов `reserve()` не меняет логического содержимого контейнера.

```
// Занести в coll копию содержимого l
coll.assign(l.begin(),l.end());
```

Обращение к элементам

В табл. 6.5 перечислены все операции прямого обращения к элементам векторов. Как принято в С и C++, первому элементу вектора соответствует индекс 0, а последнему — индекс `size()-1`. Таким образом, *n*-му элементу соответствует индекс *n-1*. Для неконстантных векторов эти операции возвращают ссылку на элемент и поэтому могут использоваться для модификации элементов (при условии, что модификация не запрещена по другим причинам).

Таблица 6.5. Операции обращения к элементам вектора

Операция	Описание
<code>c.at(idx)</code>	Возвращает элемент с индексом <code>idx</code> (при недопустимом значении индекса генерируется исключение <code>out_of_range</code>)
<code>c[idx]</code>	Возвращает элемент с индексом <code>idx</code> (без интервальной проверки!)
<code>c.front()</code>	Возвращает первый элемент (без проверки его существования!)
<code>c.back()</code>	Возвращает последний элемент (без проверки его существования!)

Самый важный аспект для вызывающей стороны — наличие или отсутствие интервальной проверки при обращении к элементу. Такая проверка выполняется только функцией `at()`. Если индекс не входит в интервал допустимых значений, генерируется исключение `out_of_range` (см. с. 45). Остальные функции выполняются *без проверки*, и интервальные ошибки приводят к непредсказуемым последствиям. Вызов оператора `[]`, функций `front()` и `back()` для пустого контейнера всегда приводит к непредсказуемым последствиям.

```
std::vector<Elem> coll;           // Пустой вектор!
coll[5] = elem;                  // ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ
std::cout << coll.front();       // ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ
```

Следовательно, перед вызовом оператора `[]` необходимо убедиться в том, что индекс имеет допустимое значение, а перед вызовом функции `front()` или `back()` — что контейнер не пуст:

```
std::vector<Elem> coll;           // Пустой вектор!
if (coll.size() > 5) {
    coll[5] = elem;              // OK
}
if (!coll.empty()) {
    cout << coll.front();      // OK
}
coll.at(5) = elem;                // Генерирует исключение out_of_range
```

Функции получения итераторов

Векторы поддерживают стандартный набор операций для получения итераторов (табл. 6.6). Итераторы векторов относятся к категории итераторов произвольного доступа (категории итераторов рассматриваются на с. 257). Это означает, что с векторами в принципе могут использоваться все алгоритмы STL.

Таблица 6.6. Операции получения итераторов

Операция	Описание
c.begin()	Возвращает итератор произвольного доступа для первого элемента
c.end()	Возвращает итератор произвольного доступа для позиции за последним элементом
c.rbegin()	Возвращает обратный итератор для первого элемента при переборе в обратном направлении
c.rend()	Возвращает обратный итератор для позиции за последним элементом при переборе в обратном направлении

Фактический тип итераторов определяется реализацией, однако для векторов итераторы часто оформляются в виде обычных указателей. Обычный указатель соответствует требованиям к итератору произвольного доступа, а поскольку по своей внутренней структуре вектор, как правило, представляет собой массив, обычные указатели обеспечивают нужное поведение. Впрочем, рассчитывать на то, что итератор является указателем, не следует. Например, в безопасной версии STL с проверкой интервальных ошибок и других потенциальных проблем итераторы обычно оформляются в виде вспомогательного класса. На с. 264 рассказано о неприятностях, возникающих из-за различий между реализациями итераторов (в виде указателей и классов).

Итераторы остаются действительными до момента вставки или удаления элемента с меньшим индексом или перераспределения памяти с изменением емкости (см. с. 157).

Вставка и удаление элементов

В табл. 6.7 перечислены операции вставки и удаления элементов, поддерживаемые векторами. Как это обычно бывает при использовании STL, правильность аргументов обеспечивается вызывающей стороной. Итераторы должны ссылаться на правильные позиции, конец интервала не должен предшествовать началу, элементы не должны удаляться из пустого контейнера.

Таблица 6.7. Операции вставки и удаления для векторов

Операция	Описание
c.insert(pos, elem)	Вставляет в позицию итератора pos копию элемента elem и возвращает позицию нового элемента
c.insert(pos, n, elem)	Вставляет в позицию итератора pos n копий элемента elem (и не возвращает значения)

Операция	Описание
c.insert(pos,beg,end)	Вставляет копию всех элементов интервала [beg,end) в позицию итератора pos (и не возвращает значения)
c.push_back(elem)	Присоединяет копию elem в конец вектора
c.pop_back()	Удаляет последний элемент (не возвращая его)
c.erase(pos)	Удаляет элемент в позиции итератора pos и возвращает позицию следующего элемента
c.erase(beg,end)	Удаляет все элементы из интервала [beg,end) и возвращает позицию следующего элемента
c.resize(num)	Приводит контейнер к размеру num (если size() при этом увеличивается, новые элементы создаются своим конструктором по умолчанию)
c.resize(num,elem)	Приводит контейнер к размеру num (если size() при этом увеличивается, новые элементы создаются как копии elem)
c.clear()	Удаляет все элементы (контейнер остается пустым)

Что касается эффективности, следует помнить, что вставка и удаление выполняются быстрее, если:

- элементы вставляются или удаляются в конце интервала;
- на момент вызова емкость контейнера достаточно велика, чтобы операция выполнялась без перераспределения памяти;
- группа элементов обрабатывается одним вызовом вместо нескольких последовательных вызовов.

Вставка и удаление элементов приводят к появлению недействительных ссылок, указателей и итераторов, ссылающихся на элементы после позиции вставки. Если вставка приводит к перераспределению памяти, то недействительными становятся все ссылки, итераторы и указатели.¹

Векторы не поддерживают операции прямого удаления элементов с некоторым значением. Для этой цели применяется алгоритм. Например, следующая команда удаляет все элементы со значением val:

```
std::vector<Elem> coll;
...
// Удаление всех элементов со значением val
coll.erase(remove(coll.begin(),coll.end(),
                  val),
            coll.end());
```

Эта конструкция описана на с. 125.

Следующий фрагмент удаляет из вектора только первый элемент с некоторым значением:

```
std::vector<Elem> coll;
...
```

¹ Естественно, позиция в операциях вставки/удаления задается итератором. — Примеч. перев.

```
// Удаление первого элемента со значением val
std::vector<Elem>::iterator pos;
pos = find(coll.begin(), coll.end(),
           val);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

Векторы как обычные массивы

В спецификации стандартной библиотеки C++ не сказано, что элементы вектора должны храниться в непрерывном блоке памяти. Тем не менее подразумевалось, что такая реализация гарантирована, а соответствующие изменения будут внесены в спецификацию. Следовательно, для любого действительного индекса *i* в векторе *v* заведомо истинно следующее условие:

```
&v[i] == &v[0] + i
```

Из гарантированного выполнения этого условия следует один важный факт: вектор может задействоваться во всех случаях, когда в программе используется динамический массив. Например, в векторе можно хранить данные обычных строк С типа *char** или *const char**:

```
std::vector<char> v;           // Создание вектора как
                                // динамического массива типа char

v.resize(41);                  // Выделить память для 41 символа (включая \0)
strcpy(&v[0], "hello, world"); // Копирование строки С в вектор
printf("%s\n", &v[0]);        // Вывод содержимого вектора в виде строки С
```

Конечно, при таком использовании вектора необходима осторожность (впрочем, при работе с динамическими массивами она необходима всегда). Например, вы должны следить за тем, чтобы размер вектора был достаточным для хранения всех копируемых данных, а если содержимое вектора потребуется как строка С, оно должно завершаться элементом \0. Однако приведенный пример показывает, что когда в программе необходим массив типа Т (например, для существующей библиотеки С), вы можете использовать вектор *vector[T]* и передать адрес первого элемента.

Обратите внимание: передавать итератор вместо адреса первого элемента было бы неправильно. Тип векторного итератора определяется реализацией; вполне возможно, что он не имеет ничего общего с обычным указателем:

```
printf("%s\n", v.begin());     // ОШИБКА (может работать,
                                // но нарушает переносимость)
printf("%s\n", &v[0]);         // OK
```

Обработка исключений

Проверка логических ошибок в векторах сведена к минимуму. Согласно стандарту, исключения генерирует только одна функция *at()* — безопасная версия

оператора индексирования (см. с. 161). Кроме того, стандарт требует, чтобы про-исходили только стандартные исключения — такие, как `bad_alloc` при нехватке памяти, или исключения при выполнении пользовательских операций.

Если функции, вызванные вектором (функции, определенные при определении типа элемента или переданные пользователем), инициируют исключения, стандартная библиотека C++ гарантирует следующее.

- Если исключение происходит при вставке элемента функцией `push_back()`, эта функция не вносит изменений в контейнер.
- Если операции копирования (копирующий конструктор и оператор присваивания) не генерируют исключений, то функция `insert()` либо выполняется успешно, либо не вносит изменений.
- Функция `pop_back()` не генерирует исключений.
- Если операции копирования (копирующий конструктор и оператор присваивания) не генерируют исключений, то функции `erase()` и `clear()` тоже не генерируют исключений.
- Функция `swap()` не генерирует исключений.
- Если используемые элементы не генерируют исключений во время операций копирования (копирующий конструктор и оператор присваивания), то любая операция либо выполняется успешно, либо не вносит изменений в контейнер. Такими элементами могут быть «обычные данные», то есть типы, не использующие специальные возможности C++. Например, любая простая структура С относится к «обычным данным».

Все перечисленные гарантии основаны на том, что деструкторы не генерируют исключения. На с. 148 приведены общие сведения об обработке исключений в STL, а на с. 254 перечислены все контейнерные операции, для которых предоставляются особые гарантии в отношении исключений.

Примеры использования векторов

Ниже приведен простой пример, демонстрирующий практическое применение векторов.

```
// cont/vector1.cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
    // Создание пустого вектора для хранения строк
    vector<string> sentence;

    // Резервируем память для пяти элементов.
    // чтобы предотвратить перераспределение
    sentence.reserve(5);
```

```
// Присоединение нескольких элементов
sentence.push_back("Hello,");
sentence.push_back("how");
sentence.push_back("are");
sentence.push_back("you");
sentence.push_back("?");
cout << endl;

// Вывод элементов, разделенных пробелами
copy(sentence.begin(), sentence.end(),
      ostream_iterator<string>(cout, " "));
cout << endl;

// Вывод "служебных данных"
cout << " max_size(): " << sentence.max_size() << endl;
cout << " size(): " << sentence.size() << endl;
cout << " capacity(): " << sentence.capacity() << endl;

// Перестановка второго и четвертого элемента
swap(sentence[1], sentence[3]);

// Вставка элемента "always" перед элементом "?"
sentence.insert(find(sentence.begin(), sentence.end(), "?"),
                 "always");

// Присваивание "!" последнему элементу
sentence.back() = "!";

// Вывод элементов, разделенных пробелами
copy(sentence.begin(), sentence.end(),
      ostream_iterator<string>(cout, " "));
cout << endl;

// Повторный вывод "служебных данных"
cout << " max_size(): " << sentence.max_size() << endl;
cout << " size(): " << sentence.size() << endl;
cout << " capacity(): " << sentence.capacity();
}
```

Результат выполнения программы выглядит примерно так:

```
Hello, how are you?
max_size(): 268435455
size(): 5
capacity(): 5
Hello, you are how always !
max_size(): 268435455
size(): 6
capacity(): 10
```

Обратите внимание на формулировку «примерно так». Значения `max_size()` и `capacity()` зависят от реализации. Например, в данном случае видно, что если вектор не помещается в зарезервированной памяти, реализация удваивает его емкость.

Класс `vector<bool>`

Для векторов, содержащих элементы логического типа, в стандартной библиотеке C++ определена специализированная разновидность класса `vector`. Это было сделано для того, чтобы оптимизированная версия занимала меньше места, чем стандартная реализация `vector` для типа `bool`. В стандартной реализации для каждого элемента резервируется минимум 1 байт. Во внутреннем представлении специализированной реализации `vector<bool>` каждый элемент обычно представляется одним битом, поэтому она занимает в восемь раз меньше памяти. Впрочем, оптимизация не дается даром: в C++ минимальное адресуемое значение должно иметь размер минимум 1 байт. Следовательно, специализированная версия требует специальной обработки ссылок и итераторов.

В результате `vector<bool>` не удовлетворяет всем требованиям других векторов (в частности, значение `vector<bool>::reference` не является полноценным l-значением, а итератор `vector<bool>::iterator` не является итератором произвольного доступа). Следовательно, код шаблона работает с векторами любого типа, за исключением `bool`. Вдобавок класс `vector<bool>` может уступать обычным реализациям по скорости работы, потому что операции с элементами приходится преобразовывать в операции с битами. Впрочем, внутреннее устройство `vector<bool>` зависит от реализации, поэтому эффективность (как скорость работы, так и затраты памяти) может быть разной.

Учтите, что класс `vector<bool>` представляет собой нечто большее, чем специализацию класса `vector<>` для `bool`. В него также включена поддержка специальных операций, упрощающих работу с флагами и наборами битов.

Размер контейнера `vector<bool>` изменяется динамически, поэтому его можно рассматривать как битовое поле с динамическим размером. Иначе говоря, вы можете добавлять и удалять биты. Если вы работаете с битовым полем, имеющим статический размер, вместо класса `vector<bool>` лучше использовать `bitset`. Класс `bitset` рассматривается на с. 444.

Дополнительные операции контейнера `vector<bool>` перечислены в табл. 6.8. Операция `flip()`, производящая логическую инверсию, может применяться как ко всем битам, так и кциальному биту вектора.

Таблица 6.8. Специальные операции `vector<bool>`

Операция	Описание
<code>c.flip()</code>	Инвертирует все логические элементы
<code>m[idx].flip()</code>	Инвертирует логический элемент с заданным индексом
<code>m[idx] = val</code>	Присваивает значение логическому элементу с индексом <code>idx</code> (присваивание одного бита)
<code>m[idx1] = m[idx2]</code>	Присваивает значение элемента с индексом <code>idx2</code> элементу с индексом <code>idx1</code>

Обратите внимание на возможность вызова `flip()` для одного логического элемента. На первый взгляд это выглядит странно, потому что оператор индексирования возвращает значение типа `bool`, а вызов `flip()` для этого базового типа невозможен. В данном случае класс `vector<bool>` использует стандартную методику, основанную на применении так называемых *заместителей*: для `vector<bool>` возвращаемое значение оператора индексирования (и других операторов, возвращающих элементы) оформлено в виде вспомогательного класса. Если вы хотите, чтобы возвращаемое значение интерпретировалось как значение типа `bool`, используется автоматическое преобразование типа. Для других операций определяются функции класса. Соответствующая часть объявления `vector<bool>` выглядит так:

```
namespace std {
    class vector<bool> {
        public:
            // Вспомогательный тип для оператора индексирования
            class reference {
                ...
                public:
                    // Автоматическое преобразование типа к bool
                    operator bool() const;

                    // Присваивание
                    reference& operator=(const bool);
                    reference& operator=(const reference&);

                    // Инверсия бита
                    void flip();
            };
            ...
            // Операции обращения к элементам
            // - возвращается тип reference вместо bool
            reference operator[](size_type n);
            reference at(size_type n);
            reference front();
            reference back();
            ...
    };
}
```

Как видно из этого фрагмента, все функции обращения к элементам возвращают тип `reference`. Следовательно, вы также можете использовать следующие команды:

```
c.front().flip(); // Инверсия первого логического элемента
c.at(5) = c.back(); // Присвоить последний элемент элементу с индексом 5
```

Как обычно,зывающая сторона должна проследить за тем, чтобы в векторе существовали первый, последний и шестой элементы.

Внутренний тип `reference` используется только для неконстантных контейнеров типа `vector<bool>`. Константные функции обращения к элементам возвращают обычные значения типа `bool`.

Деки

Дек очень похож на вектор. Он тоже работает с элементами, оформленными в динамический массив, поддерживает произвольный доступ и обладает практически тем же интерфейсом. Различие заключается в том, что динамический массив дека открыт с обоих концов. По этой причине дек быстро выполняет операции вставки и удаления как с конца, так и с начала (рис. 6.2).

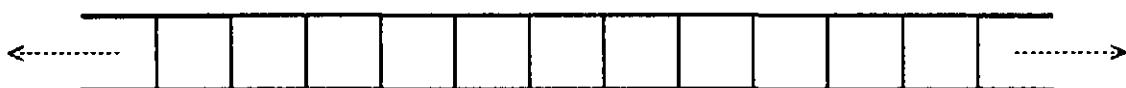


Рис. 6.2. Логическая структура дека

Дек обычно реализуется в виде набора блоков; первый и последний блоки наращиваются в противоположных направлениях (рис. 6.3).

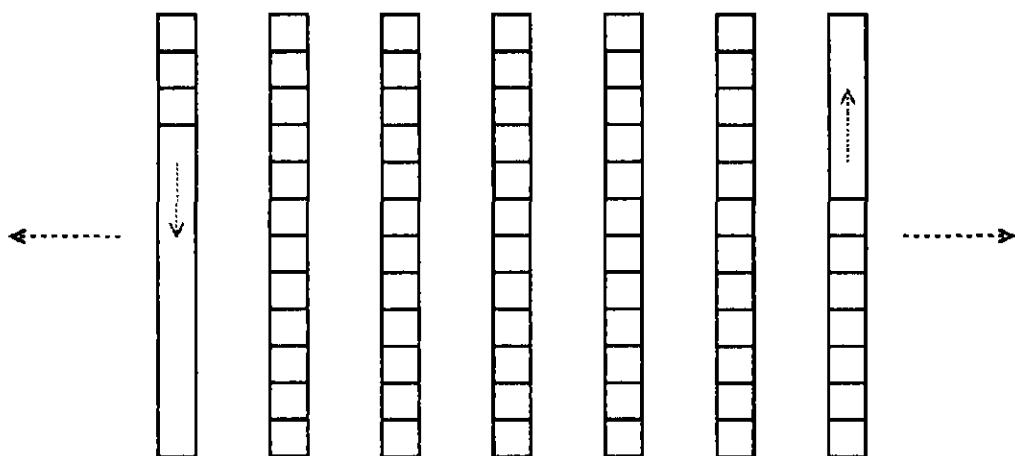


Рис. 6.3. Внутренняя структура дека

Чтобы использовать дек в программе, необходимо включить в нее заголовочный файл `<deque>`¹:

```
#include <deque>
```

Тип дека определяется как шаблон класса в пространстве имен `std`:

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

¹ В исходной версии STL дек определялся в заголовочном файле `<deque.h>`.

Как и в случае с вектором, тип элементов дека передается в первом параметре шаблона. Допускается любой тип, поддерживающий присваивание и копирование. Необязательный второй параметр шаблона определяет модель памяти, по умолчанию используется модель allocator (см. главу 15)¹.

Возможности деков

По своим возможностям деки во многом отличаются от векторов.

- Вставка и удаление выполняются быстро как в начале, так и в конце (для векторов эти операции выполняются быстро только в конце). Операции выполняются с амортизированным постоянным временем.
- Внутренняя структура содержит дополнительный уровень ссылок, поэтому обращение к элементам и перемещение итератора в деках обычно выполняются чуть медленнее.
- Итераторы должны быть умными указателями особого типа. Обычные указатели не подходят из-за необходимости перехода между блоками.
- В системах с ограниченными размерами блоков памяти (например, в некоторых системах РС) дек может содержать больше элементов, поскольку он не ограничивается одним блоком памяти. Следовательно, функция `max_size()` может возвращать для деков большую величину.
- Деки не позволяют управлять емкостью и моментом перераспределения памяти. В частности, при любых операциях вставки и удаления, выполняемых не в начале или конце вектора, становятся недействительными все указатели, ссылки и итераторы, ссылающиеся на элементы дека. Однако перераспределение памяти в общем случае выполняется более эффективно, чем для векторов, потому что из-за особенностей своей внутренней структуры декам не приходится копировать все элементы.
- Освобождение неиспользуемых блоков может привести к уменьшению объема памяти, занимаемой деком (хотя как это происходит и происходит ли вообще, зависит от реализации).

Следующие особенности векторов характерны также и для деков.

- Вставка и удаление элементов в середине контейнера выполняется относительно медленно, потому что для освобождения места или заполнения пропуска приходится перемещать элементы с обоих концов.
- Итераторы являются итераторами произвольного доступа.
Подведем итог. Выбирайте дек, если выполняются следующие условия:
- вставка элементов выполняется с обоих концов (классический пример очереди);
- в программе не используются ссылки на элементы контейнера;
- важно, чтобы контейнер освобождал неиспользуемую память (хотя стандарт таких гарантий не дает).

¹ В системах, не поддерживающих значения по умолчанию для параметров шаблонов, второй параметр обычно отсутствует.

Интерфейс векторов и деков почти не отличается. Если в программе не используются специфические возможности вектора или дека, вы можете легко опробовать оба контейнера.

Операции над деками

В табл. 6.9–6.11 перечислены все операции, поддерживаемые деками.

Таблица 6.9. Конструкторы и деструктор деков

Операция	Описание
<code>deque<Elem> c</code>	Создает пустой дек, не содержащий ни одного элемента
<code>deque<Elem> c1(c2)</code>	Создает копию другого дека того же типа (с копированием всех элементов)
<code>deque<Elem> c(n)</code>	Создает дек из n элементов, создаваемых конструктором по умолчанию
<code>deque<Elem> c(n, elem)</code>	Создает дек, инициализируемый n копиями элемента elem
<code>deque<Elem> c(beg,end)</code>	Создает дек, инициализируемый элементами интервала [beg,end)
<code>c~deque<Elem>()</code>	Уничтожает все элементы и освобождает память

Таблица 6.10. Немодифицирующие операции над деками

Операция	Описание
<code>c.size()</code>	Возвращает фактическое количество элементов
<code>c.empty()</code>	Проверяет, пуст ли контейнер (эквивалент <code>size() == 0</code> , но иногда выполняется быстрее)
<code>c.max_size()</code>	Возвращает максимально возможное количество элементов
<code>c1 == c2</code>	Проверяет равенство c1 и c2
<code>c1 != c2</code>	Проверяет неравенство c1 и c2 (эквивалент !(c1 == c2))
<code>c1 < c2</code>	Проверяет, что c1 меньше c2
<code>c1 > c2</code>	Проверяет, что c1 больше c2 (эквивалент c2 < c1)
<code>c1 <= c2</code>	Проверяет, что c1 не больше c2 (эквивалент !(c2 < c1))
<code>c1 >= c2</code>	Проверяет, что c1 не меньше c2 (эквивалент !(c1 < c2))
<code>c.at(idx)</code>	Возвращает элемент с индексом idx (при недопустимом значении индекса генерируется исключение <code>out_of_range</code>)
<code>c[idx]</code>	Возвращает элемент с индексом idx (без интервальной проверки!)
<code>c.front()</code>	Возвращает первый элемент (без проверки его существования!)
<code>c.back()</code>	Возвращает последний элемент (без проверки его существования!)
<code>c.rbegin()</code>	Возвращает обратный итератор для первого элемента при переборе в обратном направлении
<code>c.rend()</code>	Возвращает обратный итератор для позиции за последним элементом при переборе в обратном направлении

Таблица 6.11. Модифицирующие операции над деками

Операция	Описание
<code>c1 = c2</code>	Присваивает <code>c1</code> все элементы <code>c2</code>
<code>c.assign(n, elem)</code>	Присваивает <code>n</code> копий элемента <code>elem</code>
<code>c.assign(beg, end)</code>	Присваивает элементы интервала <code>[beg, end)</code>
<code>c1.swap(c2)</code>	Меняет местами содержимое <code>c1</code> и <code>c2</code>
<code>swap(c1, c2)</code>	То же, но в форме глобальной функции
<code>c.insert(pos, elem)</code>	Вставляет копию <code>elem</code> в позицию итератора <code>pos</code> и возвращает позицию нового элемента
<code>c.insert(pos, n, elem)</code>	Вставляет <code>n</code> копий <code>elem</code> в позицию итератора <code>pos</code> (и не возвращает значения)
<code>c.insert(pos, beg, end)</code>	Вставляет в позицию итератора <code>pos</code> копию всех элементов интервала <code>[beg, end)</code> (и не возвращает значения)
<code>c.push_back(elem)</code>	Присоединяет копию <code>elem</code> в конец дека
<code>c.pop_back()</code>	Удаляет последний элемент (не возвращая его)
<code>c.push_front(elem)</code>	Вставляет копию <code>elem</code> в начало дека
<code>c.pop_front()</code>	Удаляет первый элемент (не возвращая его)
<code>c.erase(pos)</code>	Удаляет элемент в позиции итератора <code>pos</code> и возвращает позицию следующего элемента
<code>c.erase(beg, end)</code>	Удаляет все элементы из интервала <code>[beg, end)</code> и возвращает позицию следующего элемента
<code>c.resize(num)</code>	Приводит контейнер к размеру <code>num</code> (если <code>size()</code> при этом увеличивается, новые элементы создаются своим конструктором по умолчанию)
<code>c.resize(num, elem)</code>	Приводит контейнер к размеру <code>num</code> (если <code>size()</code> при этом увеличивается, новые элементы создаются как копии <code>elem</code>)
<code>c.clear()</code>	Удаляет все элементы (контейнер остается пустым)

Операции деков отличаются от операций векторов только в следующих отношениях:

- деки не поддерживают функции, связанные с емкостью (`capacity()` и `reserve()`);
- в деках определены прямые функции вставки и удаления первого элемента (`push_front()` и `pop_back()`).

Поскольку остальные операции остались без изменений, здесь они не рассматриваются. За описаниями обращайтесь на с. 159.

При работе с деками необходимо учитывать следующее.

- Функции обращения к элементам (кроме `at()`) не проверяют правильность индексов и итераторов.
- Вставка или удаление элементов может привести к перераспределению памяти. Это означает, что в результате любой вставки или удаления могут стать недействительными все указатели, ссылки и итераторы, ссылающиеся

на другие элементы дека. Данное утверждение не относится к вставке элементов в начало или конец дека — в этом случае ссылки и указатели на элементы (но не итераторы!) остаются действительными.

Обработка исключений

В принципе возможности обработки исключений в деках аналогичны соответствующим возможностям векторов (см. с. 164). Дополнительные операции `push_front()` и `pop_front()` ведут себя аналогично операциям `push_back()` и `pop_back()` соответственно. Таким образом, стандартная библиотека C++ обеспечивает следующие гарантии:

- если при вставке элемента функцией `push_back()` или `push_front()` происходит исключение, эти функции не вносят изменений;
- функции `pop_back()` и `pop_front()` не генерируют исключений.

На с. 148 приведены общие сведения об обработке исключений в STL, а на с. 254 перечислены все контейнерные операции, для которых предоставляются особые гарантии в отношении исключений.

Примеры использования деков

Ниже приводится простой пример, демонстрирующий практическое применение деков.

```
// cont/deque1.cpp
#include <iostream>
#include <deque>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
    // Создание пустого дека для хранения строк
    deque<string> coll;
    \

    // Вставка нескольких элементов
    coll.assign(3, string("string"));
    coll.push_back("last string");
    coll.push_front("first string");

    // Вывод элементов с разделением символом новой строки
    copy(coll.begin(), coll.end(),
        ostream_iterator<string>(cout, "\n"));
    cout << endl;

    // Удаление первого и последнего элементов
```

```

coll.pop_front();
coll.pop_back();

// Вставка "another" во все элементы, кроме первого
for (unsigned i=1; i<coll.size(); ++i) {
    coll[i] = "another " + coll[i];
}

// Увеличение размера до четырех элементов
coll.resize (4, "resized string");

// Вывод элементов с разделением символом новой строки
copy (coll.begin(), coll.end(),
      ostream_iterator<string>(cout, "\n"));
}

```

Результат выполнения программы выглядит так:

```

first string
string
string
string
last string

string
another string
another string
resized string

```

Списки

Элементы контейнера `list` библиотеки STL объединены в двусвязный список (рис. 6.4). Как обычно, эта реализация не указана в спецификации стандартной библиотеки C++, но обусловлена ограничениями и требованиями, предъявляемыми к контейнеру этого типа.

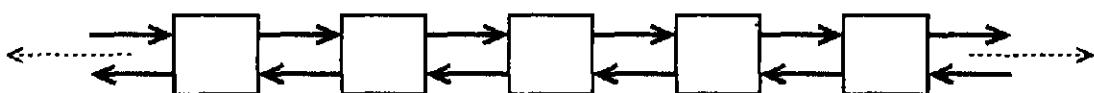


Рис. 6.4. Структура списка

Чтобы использовать список в программе, необходимо включить в нее заголовочный файл `<list>`¹:

```
#include <vector>
```

¹ В исходной версии STL список определялся в заголовочном файле `<list.h>`.

Тип списка определяется как шаблон класса в пространстве имен std:

```
namespace std {
    template <class T,
              class Allocator = allocator<T> >
    class list;
}
```

Элементы списка относятся к произвольному типу T, поддерживающему присваивание и копирование. Необязательный второй параметр шаблона определяет модель памяти (см. главу 15). По умолчанию используется модель allocator, определенная в стандартной библиотеке C++¹.

Возможности списков

По своей внутренней структуре списки полностью отличаются от векторов и деков. Важнейшие различия перечислены ниже.

- Список не поддерживает произвольный доступ к элементам. Например, чтобы обратиться к пятому элементу, необходимо перебрать первые четыре элемента по цепочке ссылок. Это означает, что обращение к произвольному элементу списка выполняется относительно медленно.
- Вставка и удаление элементов в любой позиции выполняются быстро, причем не только с концов контейнера. Вставка и удаление всегда выполняются с постоянным временем, поскольку не требуют перемещения других элементов. Во внутренней реализации изменяются только значения нескольких указателей.
- В результате вставки и удаления элементов указатели, ссылки и итераторы, относящиеся к другим элементам, остаются действительными.
- Обработка исключений в списках реализована так, что практически каждая операция завершается успешно или не вносит изменений. Иначе говоря, список не может оказаться в промежуточном состоянии, в котором операция завершена только наполовину.

Эти принципиальные различия отражены в функциях списков, что делает их непохожими на функции векторов и деков.

- Поскольку списки не поддерживают произвольный доступ к элементам, в них не определены ни оператор индексирования, ни функция at().
- Списки не поддерживают операции, связанные с емкостью и перераспределением памяти, потому что такие операции просто не нужны. У каждого элемента имеется собственная память, которая остается действительной до удаления элемента.
- Списки поддерживают много специальных функций для перемещения элементов. Эти функции класса представляют собой оптимизированные версии одноименных универсальных алгоритмов. Оптимизация основана на замене указателей вместо копирования и перемещения значений.

¹ В системах, не поддерживающих значения по умолчанию для параметров шаблонов, второй параметр обычно отсутствует.

Операции над списками

Операции создания, копирования и уничтожения

Список поддерживает такой же набор операций создания, копирования и уничтожения, как и все остальные последовательные контейнеры. Конструкторы и деструкторы списков перечислены в табл. 6.12. Некоторые из возможных источников инициализации упоминаются на с. 154.

Таблица 6.12. Конструкторы и деструкторы списков

Операция	Описание
<code>list<Elem> c</code>	Создает пустой список, не содержащий ни одного элемента
<code>list<Elem> c1(c2)</code>	Создает копию другого списка того же типа (с копированием всех элементов)
<code>list<Elem> c(n)</code>	Создает список из n элементов, создаваемых конструктором по умолчанию
<code>list<Elem> c(n, elem)</code>	Создает список, инициализируемый n копиями элемента elem
<code>list<Elem> c(beg,end)</code>	Создает список, инициализируемый элементами интервала [beg,end)
<code>c~list<Elem>()</code>	Уничтожает все элементы и освобождает память

Немодифицирующие операции над списками

Списки поддерживают стандартный набор операций для определения размера и сравнения. Эти операции перечислены в табл. 6.13, а подробные описания отдельных операций можно найти на с. 153.

Таблица 6.13. Немодифицирующие операции над списками

Операция	Описание
<code>c.size()</code>	Возвращает фактическое количество элементов
<code>c.empty()</code>	Проверяет, пуст ли контейнер (эквивалент <code>size() == 0</code> , но иногда выполняется быстрее)
<code>c.max_size()</code>	Возвращает максимально возможное количество элементов
<code>c1 == c2</code>	Проверяет равенство c1 и c2
<code>c1 != c2</code>	Проверяет неравенство c1 и c2 (эквивалент <code>!(c1 == c2)</code>)
<code>c1 < c2</code>	Проверяет, что c1 меньше c2
<code>c1 > c2</code>	Проверяет, что c1 больше c2 (эквивалент <code>c2 < c1</code>)
<code>c1 <= c2</code>	Проверяет, что c1 не больше c2 (эквивалент <code>!(c2 < c1)</code>)
<code>c1 >= c2</code>	Проверяет, что c1 не меньше c2 (эквивалент <code>!(c1 < c2)</code>)

Присваивание

Списки также поддерживают обычный набор операций присваивания для последовательных контейнеров (табл. 6.14).

Таблица 6.14. Операции присваивания для списков

Операция	Описание
c1 = c2	Присваивает c1 все элементы c2
c.assign(n, elem)	Присваивает n копий элемента elem
c.assign(beg, end)	Присваивает элементы интервала [beg, end)
c1.swap(c2)	Меняет местами содержимое c1 и c2
swap(c1, c2)	То же, но в форме глобальной функции

Как обычно, операции присваивания соответствуют конструкторам с различными источниками инициализации (см. с. 153).

Обращение к элементам

Поскольку списки не поддерживают произвольный доступ к элементам, для прямого обращения к элементам в них предусмотрены функции `front()` и `back()`, представленные в табл. 6.15.

Таблица 6.15. Операции прямого обращения к элементам списка

Операция	Описание
<code>c.front()</code>	Возвращает первый элемент (без проверки его существования!)
<code>c.back()</code>	Возвращает последний элемент (без проверки его существования!)

Как и в предыдущих случаях, эти операции *не проверяют* наличие элементов в контейнере. Если контейнер пуст, их вызов приводит к непредсказуемым последствиям. Следовательно, перед вызовом функций необходимо убедиться в том, что контейнер содержит хотя бы один элемент. Пример:

```
std::list<Elem> coll;           // Пустой список!
std::cout << coll.front();      // ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ
if (!coll.empty()) {
    std::cout << coll.back();   // OK
}
```

Функции получения итераторов

Обращение к произвольному элементу списка возможно только с применением итераторов. Списки поддерживают стандартные функции получения итераторов (табл. 6.16). Тем не менее из-за отсутствия произвольного доступа в списках эти итераторы являются всего лишь двунаправленными. Это означает, что вы не можете вызывать алгоритмы, для работы которых необходимы итераторы произвольного доступа. К этой категории относятся все алгоритмы, заметно изменяющие порядок следования элементов (особенно алгоритмы сортировки). Тем не менее для сортировки элементов в списках существует специальная функция `sort()` (см. с. 252).

Таблица 6.16. Операции получения итераторов

Операция	Описание
c.begin()	Возвращает двунаправленный итератор для первого элемента
c.end()	Возвращает двунаправленный итератор для позиции за последним элементом
c.rbegin()	Возвращает обратный итератор для первого элемента при переборе в обратном направлении
c.rend()	Возвращает обратный итератор для позиции за последним элементом при переборе в обратном направлении

Вставка и удаление элементов

В табл. 6.17 перечислены операции вставки и удаления элементов в списках. Списки поддерживают все функции деков, а также специальные реализации алгоритмов `remove()` и `remove_if()`. Как это обычно бывает при использовании STL, правильность аргументов обеспечивается вызывающей стороной. Итераторы должны ссылаться на правильные позиции, конец интервала не должен предшествовать началу, элементы не должны удаляться из пустого контейнера.

Таблица 6.17. Операции вставки и удаления для списков

Операция	Описание
<code>c.insert(pos, elem)</code>	Вставляет копию <code>elem</code> в позицию итератора <code>pos</code> и возвращает позицию нового элемента
<code>c.insert(pos,n,elem)</code>	Вставляет <code>n</code> копий <code>elem</code> в позицию итератора <code>pos</code> (и не возвращает значения)
<code>c.insert(pos,beg,end)</code>	Вставляет копию всех элементов интервала <code>[beg,end)</code> в позицию итератора <code>pos</code> (и не возвращает значения)
<code>c.push_back(elem)</code>	Присоединяет копию <code>elem</code> в конец списка
<code>c.pop_back()</code>	Удаляет последний элемент (не возвращая его)
<code>c.push_front(elem)</code>	Вставляет копию <code>elem</code> в начало списка
<code>c.pop_front()</code>	Удаляет первый элемент (не возвращая его)
<code>c.remove(val)</code>	Удаляет все элементы со значением <code>val</code>
<code>c.remove_if(op)</code>	Удаляет все элементы, для которых <code>op(elem)</code> возвращает <code>true</code>
<code>c.erase(pos)</code>	Удаляет элемент в позиции итератора <code>pos</code> и возвращает позицию следующего элемента
<code>c.erase(beg,end)</code>	Удаляет все элементы из интервала <code>[beg,end)</code> и возвращает позицию следующего элемента
<code>c.resize(num)</code>	Приводит контейнер к размеру <code>num</code> (если <code>size()</code> при этом увеличивается, новые элементы создаются своим конструктором по умолчанию)
<code>c.resize(num,elem)</code>	Приводит контейнер к размеру <code>num</code> (если <code>size()</code> при этом увеличивается, новые элементы создаются как копии <code>elem</code>)
<code>c.clear()</code>	Удаляет все элементы (контейнер остается пустым)

Вставка и удаление выполняются быстрее, если группа элементов обрабатывается одним вызовом вместо нескольких последовательных вызовов.

Для удаления элементов в списках предусмотрены специализированные версии алгоритмов `remove()` (см. с. 371). Эти функции работают быстрее алгоритмов `remove()`, потому что используют вместо элементов только внутренние указатели. Следовательно, в отличие от векторов или деков операцию `remove()` для списков следует вызывать в форме функции класса, а не алгоритма (см. с. 163). Чтобы удалить все элементы с заданным значением, воспользуйтесь следующей конструкцией (за подробностями обращайтесь на с. 127):

```
std::list<Elem> coll;
...
// Удаление всех элементов со значением val
coll.remove(val);
```

Однако для того чтобы удалить только первый экземпляр искомого значения, придется воспользоваться алгоритмом (по аналогии с тем, как показано на с. 163 для векторов).

Функция `remove_if` позволяет определить критерий удаления элементов в виде функции или объекта функции¹. Она удаляет каждый элемент, для которого передаваемая операция возвращает `true`. Пример использования `remove_if()` для удаления всех элементов с четными значениями:

```
list.remove_if (not1(bind2nd(modulus<int>(),2)));
```

Если команда кажется непонятной, не огорчайтесь. Подробности приводятся на с. 306, а дополнительные примеры использования функций `remove()` и `remove_if()` можно найти на с. 372.

Функции врезки

Как отмечалось выше, одним из достоинств связанных списков является возможность удаления и вставки элементов в произвольной позиции с постоянным временем. При перемещении элементов из одного контейнера в другой это достоинство проявляется еще сильнее, потому что операция выполняется переназначением нескольких внутренних указателей (рис. 6.5).

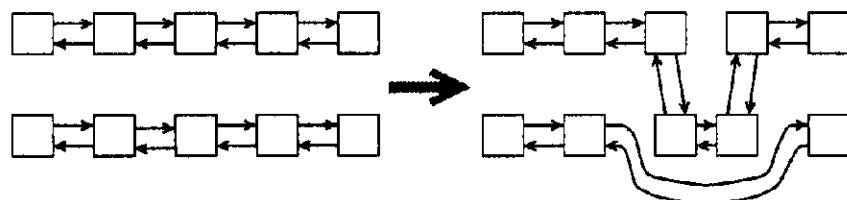


Рис. 6.5. Операции врезки изменяют порядок следования элементов списка

Для обеспечения этой возможности спискам необходимы дополнительные модифицирующие функции, которые бы изменили порядок следования и про-

¹ Функция `remove_if()` обычно неработоспособна в системах, не поддерживающих шаблонные функции классов.

изводили повторное связывание ссылок на элементы и интервалы. При помощи этих функций можно перемещать элементы как внутри одного списка, так и между разными списками (при условии, что эти списки имеют одинаковый тип). Перечень таких функций приводится в табл. 6.18, подробные описания — на с. 250, примеры — на с. 181.

Таблица 6.18. Специальные операции модификации списков

Операция	Описание
<code>c.unique()</code>	Удаляет дубликаты (элементы с одинаковыми значениями)
<code>c.unique(op)</code>	Удаляет дубликаты (элементы с одинаковыми значениями), для которых <code>op</code> возвращает <code>true</code>
<code>c1.splice(pos,c2)</code>	Перемещает все элементы <code>c2</code> в <code>c1</code> перед позицией итератора <code>pos</code>
<code>c1.splice(pos,c2,c2pos)</code>	Перемещает все элементы <code>c2</code> , начиная с позиции итератора <code>c2pos</code> , в список <code>c1</code> перед позицией итератора <code>pos</code>
<code>c1.splice(pos,c2,c2beg,c2end)</code>	Перемещает все элементы интервала <code>[c2beg,c2end]</code> списка <code>c2</code> перед позицией итератора <code>pos</code> в список <code>c1</code> (<code>c1</code> и <code>c2</code> могут совпадать)
<code>c.sort()</code>	Сортирует все элементы оператором <code><</code>
<code>c.sort(op)</code>	Сортирует все элементы по критерию <code>op</code>
<code>c1.merge(c2)</code>	Перемещает все элементы <code>c2</code> в <code>c1</code> с сохранением сортировки (предполагается, что оба контейнера содержат отсортированные элементы)
<code>c1.merge(c2,op)</code>	Перемещает все элементы <code>c2</code> в <code>c1</code> с сохранением сортировки по <code>op()</code> (предполагается, что оба контейнера содержат элементы, отсортированные по критерию <code>op()</code>)
<code>c.reverse()</code>	Переставляет все элементы в обратном порядке

Обработка исключений

Из всех стандартных контейнеров STL списки наиболее надежны в отношении исключений. Практически все операции над списками либо завершаются успешно, либо не вносят изменений. Такая гарантия не предоставляется только операторами присваивания и функцией `sort()` (они дают лишь обычную «базовую гарантию» отсутствия утечки ресурсов и нарушения контейнерных инвариантов при возникновении исключений). Функции `merge()`, `remove()`, `remove_if()` и `unique()` предоставляют гарантии при условии, что исключения не будут генерированы при сравнении элементов оператором `==` или предикатом. Таким образом, в терминологии программирования баз данных можно сказать, что списки обладают *транзакционной безопасностью*, если не использовать операцию присваивания и функцию `sort()`, а также проследить за тем, чтобы исключения не генерировались при сравнении. В табл. 6.19 перечислены все операции, предоставляющие специальные гарантии в отношении исключений. Обработка исключений в STL более подробно рассматривается на с. 148.

Таблица 6.19. Операции над списками, предоставляющие особые гарантии в отношении исключений

Операция	Описание
push_back()	Либо завершается успешно, либо не вносит изменений
push_front()	Либо завершается успешно, либо не вносит изменений
insert()	Либо завершается успешно, либо не вносит изменений
pop_back()	Не генерирует исключений
pop_front()	Не генерирует исключений
erase()	Не генерирует исключений
clear()	Не генерирует исключений
resize()	Либо завершается успешно, либо не вносит изменений
remove()	Не генерирует исключений, если они не будут сгенерированы при сравнении элементов
remove_if()	Не генерирует исключений, если они не будут сгенерированы предикатом
unique()	Не генерирует исключений, если они не будут сгенерированы при сравнении элементов
splice()	Не генерирует исключений
merge()	Либо завершается успешно, либо не вносит изменений (если исключения не будут сгенерированы при сравнении элементов)
reverse()	Не генерирует исключений
swap()	Не генерирует исключений

Примеры использования списков

В следующем примере стоит обратить особое внимание на применение специальных функций списков.

```
// cont/list1.cpp
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

void printLists (const list<int>& l1, const list<int>& l2)
{
    cout << "list1: ";
    copy (l1.begin(), l1.end(), ostream_iterator<int>(cout, " "));
    cout << endl << "list2: ";
    copy (l2.begin(), l2.end(), ostream_iterator<int>(cout, " "));
    cout << endl << endl;
}

int main()
```

```
{  
    // Создание двух пустых списков  
    list<int> list1, list2;  
  
    // Заполнение обоих списков элементами  
    for (int i=0; i<6; ++i) {  
        list1.push_back(i);  
        list2.push_front(i);  
    }  
    printLists(list1, list2);  
  
    // Вставка всех элементов list1 перед первым элементом  
    // со значением 3 в list2  
    // - find() возвращает итератор на первый элемент со значением 3  
    list2.splice(find(list2.begin(), list2.end(), // Позиция в приемнике  
                     3),  
                 list1); // Источник  
    printLists(list1, list2);  
  
    // Перемещение первого элемента в конец  
    list2.splice(list2.end(), // Позиция в приемнике  
                 list2, // Источник  
                 list2.begin()); // Позиция в источнике  
    printLists(list1, list2);  
  
    // Сортировка второго списка, присваивание list1  
    // и удаление дубликатов  
    list2.sort();  
    list1 = list2;  
    list2.unique();  
    printLists(list1, list2);  
}  
}
```

Программа выводит следующий результат:

```
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:
```

Множества и мульти множества

Множества и мульти множества автоматически сортируют свои элементы по некоторому критерию (см. главу 5). Они отличаются только тем, что мульти множества могут содержать дубликаты, а множества — нет (рис. 6.6).

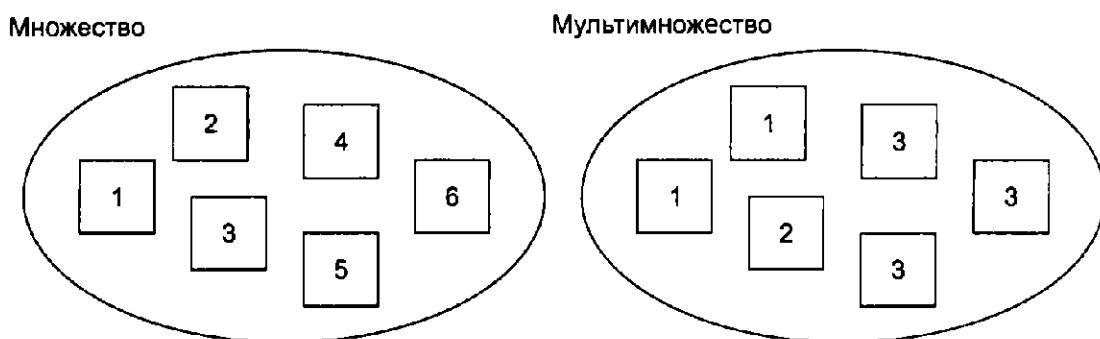


Рис. 6.6. Множество и мульти множество

Чтобы использовать множество или мульти множество в программе, необходимо включить в нее заголовочный файл `<set>`¹:

```
#include <set>
```

Типы множества и мульти множества определяются как шаблоны классов в пространстве имен `std`:

```
namespace std {
    template <class T,
              class Compare = less<T>,
              class Allocator = allocator<T> >
    class set;

    template <class T,
              class Compare = less<T>,
              class Allocator = allocator<T> >
    class multiset;
}
```

Элементы множества или мульти множества относятся к произвольному типу `T`, поддерживающему присваивание, копирование и сравнение по критерию сорти-

¹ В исходной версии STL множество определялось в заголовочном файле `<set.h>`, а мульти множество — в файле `<multiset.h>`.

ровки. Необязательный второй параметр шаблона определяет критерий сортировки. По умолчанию используется критерий `less`. Объект функции `less` сортирует элементы, сравнивая их оператором `<` (дополнительная информация об объекте `less` приводится на с. 305)¹. Необязательный третий параметр шаблона определяет модель памяти (см. главу 15). По умолчанию используется модель `allocator`, определенная в стандартной библиотеке C++².

Критерий сортировки должен определять «строгую квазиупорядоченность» (*strict weak ordering*). Строгая квазиупорядоченность определяется тремя свойствами.

○ Асимметричность.

- Для оператора `<`: если выражение $x < y$ истинно, то выражение $y < x$ ложно.
- Для предиката `op()`: если выражение `op(x,y)` истинно, то выражение `op(y,x)` ложно.

○ Транзитивность.

- Для оператора `<`: если выражения $x < y$ и $y < z$ истинны, то выражение $x < z$ истинно.
- Для предиката `op()`: если выражения `op(x,y)` и `op(y,z)` истинны, то выражение `op(x,z)` истинно.

○ Иррефлексивность.

- Для оператора `<`: выражение $x < x$ всегда ложно.
- Для предиката `op()`: выражение `op(x,x)` всегда ложно.

На основании этих свойств критерий сортировки используется также для проверки на равенство, а именно: два элемента равны, если ни один из них не меньше другого (или если ложны оба выражения `op(x,y)` и `op(y,x)`).

Возможности множеств и мульти множеств

Множества и мульти множества, как и все стандартизованные классы ассоциативных контейнеров, обычно реализуются в виде сбалансированного бинарного дерева (рис. 6.7). В стандарте такая реализация не указана, но она обусловлена требованиям к сложности операций над множествами и мульти множествами³.

Главное достоинство автоматической сортировки заключается в том, что бинарное дерево обеспечивает хорошие показатели при поиске элементов с кон-

¹ В системах, не поддерживающих значения по умолчанию для параметров шаблонов, второй параметр обычно отсутствует.

² В системах, не поддерживающих значения по умолчанию для параметров шаблонов, третий параметр обычно отсутствует.

³ Точнее говоря, множества и мульти множества обычно реализуются в виде «красно-черных деревьев». Красно-черные деревья хорошо приспособлены как для изменения количества элементов, так и для поиска. Они гарантируют, что вставка потребует не более двух изменений внутренних ссылок, а самый длинный путь к элементу будет не более чем вдвое длиннее самого короткого.

крайним значением. Функции поиска в них имеют логарифмическую сложность. Например, чтобы найти элемент в множестве или мульти множестве, содержащем 1000 элементов, процедуре поиска по дереву (функции класса) потребуется выполнить в среднем около 1/50 от количества сравнений при линейном поиске (алгоритм). За дополнительной информацией о сложности операций обращайтесь на с. 37.

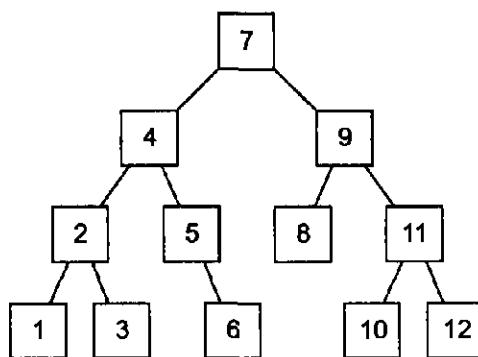


Рис. 6.7. Внутренняя структура множеств и мульти множеств

Тем не менее автоматическая сортировка также накладывает важное ограничение на множества и мульти множества: значение элементов *нельзя* изменять напрямую, потому что это может нарушить правильный порядок их расположения. Следовательно, чтобы изменить значение элемента, необходимо удалить элемент со старым значением и вставить элемент с новым значением. Этот принцип нашел свое отражение в интерфейсе:

- множества и мульти множества не поддерживают прямое обращение к элементам;
- при косвенном обращении через итераторы действует ограничение: с точки зрения итератора значение элемента является константным.

Операции над множествами и мульти множествами

Операции создания, копирования и уничтожения

В табл. 6.20 представлены конструкторы и деструктор множеств и мульти множеств.

Таблица 6.20. Конструкторы и деструктор множеств и мульти множеств

Операция	Описание
set c	Создает пустое множество или мульти множество, не содержащее ни одного элемента
set c(op)	Создает пустое множество или мульти множество, использующее критерий сортировки op
set c1(c2)	Создает копию другого множества или мульти множества того же типа (с копированием всех элементов)

продолжение 5

Таблица 6.20 (продолжение)

Операция	Описание
set c(beg,end)	Создает множество или мульти множество, инициализированное элементами интервала [beg,end)
set c(beg,end,op)	Создает множество или мульти множество с критерием сортировки op, инициализированное элементами интервала [beg,end)
c. \sim set()	Уничтожает все элементы и освобождает память

В таблице символами «set» обозначена одна из следующих конструкций:

- `set<Elem>` — множество с сортировкой по критерию `less<>` (оператор `<`);
 - `set<Elem,op>` — множество с сортировкой по критерию op;
 - `multiset<Elem>` — мульти множество с сортировкой по критерию `less<>` (оператор `<`);
 - `multiset<Elem,op>` — мульти множество с сортировкой по критерию op.
- Существуют два варианта определения критерия сортировки.

- В параметре шаблона, например¹:

```
std::set<int,std::greater<int> > coll;
```

В этом случае критерий сортировки является частью типа. Таким образом, система типов гарантирует, что объединение возможно только для контейнеров с одним критерием сортировки. Этот способ определения критерия сортировки является наиболее распространенным. Выражаясь точнее, во втором параметре передается *тип* критерия сортировки, а конкретный критерий — это объект функции, создаваемый в контейнере. Для этого конструктор контейнера вызывает конструктор по умолчанию типа критерия сортировки. Пример определения пользовательского критерия сортировки приведен на с. 296.

- В параметре конструктора. В этом варианте можно определить тип для нескольких критериев сортировки с разными значениями или состояниями этих критериев. Такой подход удобен при формировании критериев сортировки на стадии выполнения, а также при использовании различающихся критериев сортировки, которые относятся к одному типу данных. Пример приведен на с. 198.

Если критерий сортировки не указан, по умолчанию используется объект функции `less<>`, сортирующий элементы оператором `<`².

Учтите, что критерий сортировки также используется при проверке элементов на равенство. При использовании критерия сортировки по умолчанию проверка двух элементов на равенство реализуется так:

```
if (! (elem1<elem2 || elem2<elem1))
```

¹ Обратите внимание на пробел между символами `>`. Последовательность `>` воспринимается компилятором как оператор сдвига, что приводит к синтаксической ошибке.

² В системах, не поддерживающих значения по умолчанию для параметров шаблонов, критерий сортировки обычно является обязательным параметром:

```
set<int,less<int> > coll;
```

У такой реализации есть три достоинства:

- она не требует передачи дополнительного аргумента (достаточно передать всего один аргумент — критерий сортировки);
- при определении типа элемента не обязательно определять оператор `==`;
- в программе можно использовать расходящиеся определения равенства (то есть оператор `==` и проверка по критерию сортировки могут давать разные результаты), хотя это усложняет и запутывает программу.

С другой стороны, проверка на равенство в этом варианте занимает немного больше времени, поскольку для получения результата приходится выполнять два сравнения. Впрочем, если первое под выражение дает истинный результат, то второе под выражение не обрабатывается.

Полное имя типа для контейнера получается излишне сложным и громоздким, поэтому для него рекомендуется определить псевдоним (то же самое полезно сделать и для определений итераторов):

```
typedef std::set<int, std::greater<int> > IntSet;
...
IntSet coll;
IntSet::iterator pos;
```

Конструктор, которому передается начало и конец интервала, может применяться для инициализации контейнера элементами контейнеров, относящихся к другим типам (от массива до стандартного входного потока данных). За подробностями обращайтесь на с. 153.

Немодифицирующие операции над множествами и мульти множествами

Множества и мульти множества поддерживают обычный набор операций для получения размера контейнера и сравнения элементов (табл. 6.21).

Таблица 6.21. Немодифицирующие операции над множествами и мульти множествами

Операция	Описание
<code>c.size()</code>	Возвращает фактическое количество элементов
<code>c.empty()</code>	Проверяет, пуст ли контейнер (эквивалент <code>size() == 0</code> , но иногда выполняется быстрее)
<code>c.max_size()</code>	Возвращает максимально возможное количество элементов
<code>c1 == c2</code>	Проверяет равенство <code>c1</code> и <code>c2</code>
<code>c1 != c2</code>	Проверяет неравенство <code>c1</code> и <code>c2</code> (эквивалент <code>!(c1 == c2)</code>)
<code>c1 < c2</code>	Проверяет, что <code>c1</code> меньше <code>c2</code>
<code>c1 > c2</code>	Проверяет, что <code>c1</code> больше <code>c2</code> (эквивалент <code>c2 < c1</code>)
<code>c1 <= c2</code>	Проверяет, что <code>c1</code> не больше <code>c2</code> (эквивалент <code>!(c2 < c1)</code>)
<code>c1 >= c2</code>	Проверяет, что <code>c1</code> не меньше <code>c2</code> (эквивалент <code>!(c1 < c2)</code>)

Операции сравнения определены только для однотипных контейнеров. Это означает совпадение *как* типов элементов, *так и* критерия сортировки; в противном случае происходит ошибка компиляции. Пример:

```
std::set<float> c1; // Критерий сортировки: std::less<>
std::set<float> c2; // Критерий сортировки: std::greater<float>
...
if (c1 == c2) {      // ОШИБКА: разные типы
    ...
}
```

Отношение «меньше/больше» между контейнерами проверяется по лексикографическому критерию (см. с. 356). Для сравнения контейнеров разных типов (с разными критериями сортировки) необходимо использовать алгоритмы, описанные на с. 352.

Специальные операции поиска

Множества и мульти множества оптимизированы для поиска элементов, поэтому в этих контейнерах определены специальные функции поиска (табл. 6.22). Эти функции представляют собой оптимизированные версии одноименных универсальных алгоритмов. Всегда используйте оптимизированные версии функций для множеств и мульти множеств, обеспечивающие логарифмическую сложность вместо линейной сложности универсальных алгоритмов. Например, поиск в коллекции из 1000 элементов требует в среднем 10 сравнений вместо 500 (см. с. 37).

Таблица 6.22. Специальные операции поиска в множествах и мульти множествах

Операция	Описание
count(elem)	Возвращает количество элементов со значением elem
find(elem)	Возвращает позицию первого элемента со значением elem (или end())
lower_bound(elem)	Возвращает первую позицию, в которой может быть вставлен элемент elem (первый элемент \geq elem)
upper_bound(elem)	Возвращает последнюю позицию, в которой может быть вставлен элемент elem (первый элемент $>$ elem)
equal_range(elem)	Возвращает первую и последнюю позиции, в которых может быть вставлен элемент elem (интервал, в котором элементы $=$ elem)

Функция `find()` ищет первый элемент, значение которого передается в аргументе, и возвращает его позицию в виде итератора. Если поиск оказывается безуспешным, функция `find()` возвращает `end()`.

Функции `lower_bound()` и `upper_bound()` возвращают соответственно первую и последнюю позиции, в которых может быть вставлен элемент с заданным значением. Иначе говоря, `lower_bound()` возвращает позицию первого элемента, значение которого больше либо равно аргументу, а функция `upper_bound()` возвращает позицию последнего элемента, значение которого больше аргумента. Функция `equal_range()` возвращает значения `lower_bound()` и `upper_bound()` в виде объекта типа `pair` (тип `pair` описан на с. 50). Таким образом, функция возвращает интервал

элементов, значения которых равны переданному аргументу. Если `lower_bound()` или первый компонент `equal_range()` совпадает с `upper_bound()` или вторым компонентом `equal_range()`, то в множестве или мульти множестве отсутствуют элементы с заданным значением. Естественно, в множестве интервал `equal_range()` содержит не более одного элемента.

Следующий пример демонстрирует применение функций `lower_bound()`, `upper_bound()` и `equal_range()`.

```
// cont/set2.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 выполнения программы выглядит так:

```
lower_bound(3): 4
upper_bound(3): 4
equal_range (3): 4 4

lower_bound(5): 5
upper_bound(5): 5
equal_range (5): 5 6
```

Если вместо множества использовать мульти множество, результат останется прежним.

Присваивания

В множествах и мульти множествах определены только базовые операции присваивания, поддерживаемые всеми контейнерами (табл. 6.23). Дополнительная информация приведена на с. 156.

Таблица 6.23. Операции присваивания для множеств и мульти множеств

Операция	Описание
<code>c1 = c2</code>	Присваивает <code>c1</code> все элементы <code>c2</code>
<code>c1.swap(c2)</code>	Меняет местами содержимое <code>c1</code> и <code>c2</code>
<code>swap(c1,c2)</code>	То же, но в форме глобальной функции

Для выполнения операций присваивания контейнеры должны относиться к одному типу. В частности, должны совпадать типы критериев сравнения, хотя сами критерии могут различаться. Пример использования разных критериев сортировки, относящихся к одному типу, приведен на с. 198. Если критерии различаются, в результате присваивания или перестановки они также меняются местами.

Функции получения итераторов

Множества и мульти множества не обеспечивают прямого доступа к элементам, поэтому для такого доступа приходится использовать итераторы. Множества и мульти множества поддерживают стандартный набор операций для получения итераторов (табл. 6.24).

Таблица 6.24. Операции получения итераторов для множеств и мульти множеств

Операция	Описание
<code>c.begin()</code>	Возвращает двунаправленный итератор для первого элемента (элементы считаются константными)
<code>c.end()</code>	Возвращает двунаправленный итератор для позиции за последним элементом (элементы считаются константными)
<code>c.rbegin()</code>	Возвращает обратный итератор для первого элемента при переборе в обратном направлении
<code>c.rend()</code>	Возвращает обратный итератор для позиции за последним элементом при переборе в обратном направлении

Множества и мульти множества, как и все остальные классы ассоциативных контейнеров, поддерживают двунаправленные итераторы (см. с. 261). Такие итераторы не могут использоваться алгоритмами, рассчитанными на итераторы произвольного доступа (например, сортировки или случайной перестановки элементов).

Но еще более важное ограничение заключается в том, что с точки зрения итератора все элементы считаются константными. Это необходимо для того, чтобы программа не могла нарушить порядок следования элементов, изменяя их значения. Однако в результате для элементов множества или мульти множества вызов модифицирующих алгоритмов становится невозможным. Например, удаление элементов не может осуществляться алгоритмом `remove()`, потому что «удаление» в действительности сводится к перезаписи следующими элементами

(данная тема подробно обсуждается на с. 125). Элементы множеств и мульти множеств удаляются только функциями, предоставляемыми контейнером.

Вставка и удаление элементов

В табл. 6.25 перечислены операции вставки и удаления элементов в множествах и мульти множествах.

Таблица 6.25. Операции вставки и удаления для множеств и мульти множеств

Операция	Описание
c.insert(elem)	Вставляет копию elem и возвращает позицию нового элемента; для множеств также возвращается признак успешного выполнения операции
c.insert(pos,elem)	Вставляет копию elem и возвращает позицию нового элемента (pos определяет рекомендуемую позицию, с которой следует начинать поиск позиции вставляемого элемента)
c.insert(beg,end)	Вставляет копию всех элементов интервала [beg,end) (и не возвращает значения)
c.erase(elem)	Удаляет все элементы со значением elem и возвращает количество удаленных элементов
c.erase(pos)	Удаляет элемент в позиции итератора pos (не возвращает значения)
c.erase(beg,end)	Удаляет все элементы из интервала [beg,end) (не возвращает значения)
c.clear()	Удаляет все элементы (контейнер остается пустым)

Как это обычно бывает при использовании STL, правильность аргументов обеспечивается вызывающей стороной. Итераторы должны ссылаться на правильные позиции, конец интервала не должен предшествовать началу, а элементы не должны удаляться из пустого контейнера.

Вставка и удаление выполняются быстрее, если группа элементов обрабатывается одним вызовом вместо нескольких последовательных вызовов.

Обратите внимание на тип возвращаемого значения функций вставки:

○ Интерфейс множеств:

```
pair<iterator,bool> insert(const value_type& elem);
iterator           insert(iterator pos_hint,
                           const value_type& elem);
```

○ Интерфейс мульти множеств:

```
iterator           insert(const value_type& elem);
iterator           insert(iterator pos_hint,
                           const value_type& elem);
```

Различия в типах возвращаемых значений обусловлены тем, что дубликаты разрешены в мульти множествах, но запрещены в множествах. Следовательно, вставка элемента в множество может завершиться неудачей, если множество уже содержит элемент с таким же значением. Из-за этого возвращаемым значе-

нием множества является значение типа `pair` (структура `pair` рассматривается на с. 50), то есть возвращаются два значения:

- в поле `second` структуры `pair` возвращается признак успешного выполнения вставки;
- в поле `first` структуры `pair` возвращается позиция вставленного или существующего элемента.

Во всех остальных случаях функции возвращают позицию нового элемента (или существующего элемента, если множество уже содержит элемент с таким значением).

Ниже приведен пример использования этого интерфейса при вставке. В множество с вставляется новый элемент со значением 3.3:

```
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;
}
```

Если программа должна дополнительно обработать новую или старую позицию элемента, программа усложняется:

```
// Определение переменной для возвращаемого значения insert()
std::pair<std::set<float>::iterator,bool> status;

// Вставка элемента с присваиванием возвращаемого значения
status = c.insert(value);

// Обработка возвращаемого значения
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 вызова первых двух функций в этом фрагменте может выглядеть так:

```
8.9 inserted as element 4
7.7 already exists as element 3
```

Обратите внимание: типы возвращаемых значений для функций вставки с дополнительным параметром позиции не зависят от типа контейнера. Эти функции возвращают один итератор как для множеств, так и для мульти множеств. Тем не менее вызов этих функций приводит к таким же результатам, как и вызов функции без параметра позиции. Они отличаются только по скорости работы.

Функции можно передать позицию итератора, но эта позиция воспринимается только как рекомендация для оптимизации быстродействия. Если элемент вставляется сразу же за позицией, переданной в первом аргументе, сложность операции изменяется от логарифмической до амортизированной постоянной (сложность операций рассматривается на с. 37). Тот факт, что тип возвращаемого значения функций вставки с дополнительным параметром позиции не зависит от типа контейнера, как в случае функций вставки без параметра, гарантирует, что функция вставки обладает одинаковым интерфейсом для всех типов контейнеров. Этот интерфейс используется несколькими общими итераторами вставки (за подробностями обращайтесь на с. 279).

Чтобы удалить элемент с некоторым значением, достаточно вызвать функцию `erase()`:

```
std::set<Elem> coll;  
...  
// Удаление всех элементов с переданным значением  
coll.erase(value);
```

В отличие от списков функция удаления для множеств и мульти множеств называется `erase()`, а не `remove()` (см. описание функции `remove()` на с. 179). Она ведет себя иначе и возвращает количество удаленных элементов. Для множеств возвращаются только значения 0 и 1.

Если мульти множество содержит дубликаты, вам не удастся использовать функцию `erase()` для удаления только первого дубликата. Вместо этого можно воспользоваться следующим фрагментом:

```
std::multiset<Elem> coll;  
...  
// Удаление первого элемента с переданным значением  
std::multiset<Elem>::iterator pos;  
pos = coll.find (elem);  
if (pos != coll.end()) {  
    coll.erase(pos);  
}
```

Вместо алгоритма `find()` следует использовать функцию класса `find()`, потому что она работает быстрее (см. пример на с. 163).

Обратите внимание: в данном случае также существуют различия в типе возвращаемого значения, а именно: для последовательных и ассоциативных контейнеров функция `erase()` возвращает разные типы.

○ *Последовательные контейнеры* поддерживают следующие функции `erase()`:

```
iterator erase(iterator pos);  
iterator erase(iterator beg, iterator end);
```

○ *Ассоциативные контейнеры* поддерживают следующие функции `erase()`:

```
void      erase(iterator pos);  
void      erase(iterator beg, iterator end);
```

Такие различия были внесены по соображениям быстродействия. В ассоциативных контейнерах на поиск и возвращение следующего элемента потребовалось бы дополнительное время, поскольку контейнер реализован в виде бинарного дерева. С другой стороны, чтобы программный код работал с любым контейнером, возвращаемое значение приходится игнорировать.

Обработка исключений

Множества и мульти множества относятся к категории узловых контейнеров, поэтому любая неудача при конструировании нового узла просто оставляет контейнер в прежнем состоянии. Более того, поскольку деструкторы обычно не генерируют исключений, удаление узла не может завершиться неудачей.

Однако при вставке нескольких элементов из-за необходимости сохранения упорядоченности полное восстановление после исключений становится непрактичным. Поэтому для всех одноэлементных операций вставки обеспечивается транзакционная безопасность (то есть такие операции либо завершаются успешно, либо не вносят изменений). Кроме того, все многоэлементные операции удаления всегда гарантированно завершаются успешно. Если в результате копирования/присваивания критерия сравнения возможны исключения, то функция `swap()` может их генерировать.

На с. 148 приведены общие сведения об обработке исключений в STL, а на с. 254 перечислены все контейнерные операции, для которых предоставляются особые гарантии в отношении исключений.

Примеры использования множеств и мульти множеств

Следующая программа демонстрирует некоторые возможности множеств¹:

```
// cont/set1.cpp
#include <iostream>
#include <set>
using namespace std;

int main()
{
    /* Тип коллекции:
     * - дубликаты запрещены
     * - элементы типа int
     * - сортировка по убыванию
     */
    typedef set<int, greater<int> > IntSet;

    IntSet coll1;           // Пустое множество
```

¹ Определение `distance()` изменилось, поэтому в старых версиях STL приходится включать файл `distance.hpp` (см. с. 268).

```
// Вставка элементов в произвольном порядке
coll1.insert(4);
coll1.insert(3);
coll1.insert(5);
coll1.insert(1);
coll1.insert(6);
coll1.insert(2);
coll1.insert(5);

// Перебор и вывод всех элементов
IntSet::iterator pos;
for (pos = coll1.begin(); pos != coll1.end(); ++pos) {
    cout << *pos << ' ';
}
cout << endl;

// Попытка повторной вставки значения 4
// и обработка возвращаемого значения
pair<IntSet::iterator,bool> 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;
}

// Присваивание элементов другому множеству,
// упорядоченному по возрастанию
set<int> coll2(coll1.begin(),
                 coll1.end());

// Вывод всех элементов копии
copy (coll2.begin(), coll2.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;

// Удаление всех элементов до элемента со значением 3
coll2.erase (coll2.begin(), coll2.find(3));

// Удаление всех элементов со значением 5
int num;
num = coll2.erase (5);
cout << num << " element(s) removed" << endl;

// Вывод всех элементов
copy (coll2.begin(), coll2.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;
}
```

Сначала определение типа создает сокращенное имя типа для множества целых чисел, упорядоченного по убыванию:

```
typedef set<int, greater<int>> IntSet;
```

Затем мы создаем пустое множество и вставляем в него несколько элементов функцией `insert()`:

```
IntSet coll1;
```

```
coll1.insert(4);
```

```
...
```

Обратите внимание: элемент со значением 5 вставляется дважды. Вторая вставка игнорируется, поскольку дубликаты в множествах запрещены.

После вывода всех элементов программа снова пытается вставить элемент 4. На этот раз возвращаемое значение функции `insert()` обрабатывается способом, упомянутым на с. 192.

Следующая команда создает новое множество элементов типа `int`, упорядоченных по возрастанию, и инициализирует его элементами прежнего множества¹:

```
set<int> coll2(coll1.begin(), coll1.end());
```

Контейнеры используют разные критерии сортировки, поэтому их типы различаются, а прямое присваивание или сравнение невозможно. Тем не менее алгоритмы обычно способны работать с разными типами контейнеров, если типы их элементов совпадают или могут быть преобразованы друг к другу.

Следующая команда удаляет все элементы, предшествующие элементу со значением 3:

```
coll2.erase (coll2.begin(), coll2.find(3));
```

При этом элемент со значением 3 находится на открытом конце интервала и поэтому не удаляется.

Наконец, из контейнера удаляются все элементы со значением 5:

```
int num;
num = coll2.erase (5);
cout << num << " element(s) removed" << endl;
```

Результат выполнения программы выглядит так:

```
6 5 4 3 2 1
4 already exists
1 2 3 4 5 6
1 element(s) removed
3 4 6
```

¹ В приведенной команде задействованы некоторые новые возможности языка, а именно шаблоны функций класса и аргументы шаблонов по умолчанию. Если в вашей системе они не поддерживаются, команда заменяется следующим фрагментом:

```
set<int, less<int>> coll2;
copy (coll1.begin(), coll1.end(),
inserter(coll2,coll2.begin()));
```

Для мульти множеств эта программа выглядит несколько иначе и выводит другие результаты:

```
// cont/mset1.cpp
#include <iostream>
#include <set>
using namespace std;

int main()
{
    /* Тип коллекции:
     * - дубликаты разрешены
     * - элементы типа int
     * - сортировка по убыванию
     */
    typedef multiset<int,greater<int> > IntSet;

    IntSet col11;           // Пустое мульти множество

    // Вставка элементов в произвольном порядке
    col11.insert(4);
    col11.insert(3);
    col11.insert(5);
    col11.insert(1);
    col11.insert(6);
    col11.insert(2);
    col11.insert(5);

    // Перебор и вывод всех элементов
    IntSet::iterator pos;
    for (pos = col11.begin(); pos != col11.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;

    // Повторная вставка значения 4 и обработка возвращаемого значения
    IntSet::iterator ipos = col11.insert(4);
    cout << "4 inserted as element "
        << distance(col11.begin(),ipos) + 1
        << endl;

    // Присваивание элементов другому множеству.
    // упорядоченному по возрастанию
    multiset<int> col12(col11.begin(),
                         col11.end());

    // Вывод всех элементов копии
    copy (col12.begin(), col12.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;
```

```

// Удаление всех элементов до элемента со значением 3
coll2.erase (coll2.begin(), coll2.find(3));

// Удаление всех элементов со значением 5
int num;
num = coll2.erase (5);
cout << num << " element(s) removed" << endl;

// Вывод всех элементов
copy (coll2.begin(), coll2.end(),
      ostream_iterator<int>(cout, " "));
cout << endl;
}

```

Тип `set` везде заменен типом `multiset`. Кроме того, возвращаемое значение функции `insert()` обрабатывается иначе:

```

IntSet::iterator ipos = coll1.insert(4);
cout << "4 inserted as element "
    << distance(coll1.begin(), ipos) + 1
    << endl;

```

Поскольку в мульти множествах разрешены дубликаты, вставка завершится неудачей только при генерированном исключении. По этой причине в возвращаемом значении передается только итератор с позицией нового элемента.

Результат выполнения программы выглядит так:

```

6 5 5 4 3 2 1
4 already exists
1 2 3 4 4 5 5 6
2 element(s) removed
3 4 4 6

```

Пример определения критерия сортировки на стадии выполнения

Обычно критерий сортировки определяется как часть типа; для этого он либо явно передается во втором аргументе шаблона, либо по умолчанию используется критерий `less<>`. Но иногда критерий сортировки приходится определять во время выполнения программы или для одного типа данных определяются разные критерии сортировки. В таких случаях для критерия сортировки определяется специальный тип, в котором передается информация о сортировке. Следующая программа показывает, как это делается.

```

// cont/setcmp.cpp
#include <iostream>
#include <set>
#include "print.hpp"
using namespace std;

// Тип критерия сортировки
template <class T>

```

```
class RuntimeCmp {
public:
    enum cmp_mode {normal, reverse};
private:
    cmp_mode mode;
public:
    // Конструктор критерия сортировки
    // - по умолчанию используется значение normal
    RuntimeCmp (cmp_mode m=normal) : mode(m) {
    }
    // Сравнение элементов
    bool operator() (const T& t1, const T& t2) const {
        return mode == normal ? t1 < t2 : t2 < t1;
    }
    // Сравнение критериев сортировки
    bool operator==(const RuntimeCmp& rc) {
        return mode == rc.mode;
    }
};

// Тип множества, использующего данный критерий сортировки
typedef set<int,RuntimeCmp<int> > IntSet;

// Опережающее объявление
void fill (IntSet& set);

int main()
{
    // Создание, заполнение и вывод множества с обычным порядком следования
    // элементов - используется критерий сортировки по умолчанию
    IntSet coll1;
    fill(coll1);
    PRINT_ELEMENTS (coll1, "coll1: ");

    // Создание критерия сортировки с обратным порядком следования элементов
    RuntimeCmp<int> reverse_order(RuntimeCmp<int>::reverse);

    // Создание, заполнение и вывод множества
    // с обратным порядком следования элементов
    IntSet coll2(reverse_order);
    fill(coll2);
    PRINT_ELEMENTS (coll2, "coll2: ");

    // Присваивание элементов И критерия сортировки
    coll1 = coll2;
    coll1.insert(3);
    PRINT_ELEMENTS (coll1, "coll1: ");
```

```

// Просто для уверенности...
if (coll1.value_comp() == coll2.value_comp()) {
    cout << "coll1 and coll2 have same sorting criterion"
        << endl;
}
else {
    cout << "coll1 and coll2 have different sorting criterion"
        << endl;
}
}

void fill (IntSet& set)
{
    // Вставка элементов в произвольном порядке
    set.insert(4);
    set.insert(7);
    set.insert(5);
    set.insert(1);
    set.insert(6);
    set.insert(2);
    set.insert(5);
}

```

В этой программе `RuntimeCmp<>` представляет собой простой шаблон, позволяющий задать критерий сортировки для произвольного типа во время выполнения программы. Конструктор по умолчанию сортирует элементы по возрастанию, для чего используется стандартное значение `normal`. Также при вызове конструктора может быть передан аргумент `RuntimeCmp<>::reverse`, чтобы сортировка осуществлялась по убыванию.

Результат выполнения программы выглядит так:

```

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 same sorting criterion

```

Контейнеры `coll1` и `coll2` относятся к одному типу, и это обстоятельство используется, например, в функции `fill()`. Также обратите внимание на то, что оператор присваивания присваивает элементы и критерий сортировки (иначе присваивание легко приводило бы к нарушению упорядоченности).

Отображения и мультиотображения

Элементами отображений и мультиотображений являются пары «ключ/значение». Сортировка элементов производится автоматически на основании критерия сортировки, применяемого к ключу. Отображения и мультиотображения

отличаются только тем, что последние могут содержать дубликаты, а первые — нет (рис. 6.8).

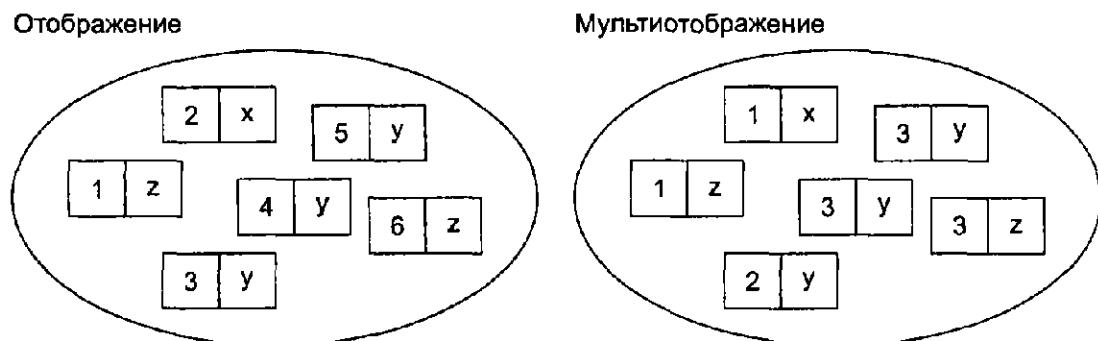


Рис. 6.8. Отображение и мультиотображение

Чтобы использовать отображение или мультиотображение в программе, необходимо включить в нее заголовочный файл `<map>`¹:

```
#include <map>
```

Типы отображения и мультиотображения определяются как шаблоны классов в пространстве имен `std`:

```
namespace std {
    template <class Key, class T,
              class Compare = less<Key>,
              class Allocator = allocator<pair<const Key, T>>
    class map;

    template <class Key, class T,
              class Compare = less<Key>,
              class Allocator = allocator<pair<const Key, T>>
    class multimap;
}
```

Первый аргумент шаблона определяет тип ключа, а второй — тип значения элемента. Элементы отображений и мультиотображений могут состоять из произвольных типов `Key` и `T`, удовлетворяющих двум условиям:

- пара «ключ/значение» должна поддерживать присваивание и копирование;
- ключ должен быть совместим с критерием сортировки.

Необязательный третий аргумент шаблона задает критерий сортировки. Как и в случае с множествами, критерий сортировки должен определять «строгую квазиупорядоченность» (см. с. 184). Элементы сортируются по ключам, значения элементов не влияют на порядок следования. Критерий сортировки также используется для проверки на равенство (два элемента равны, если ни один из них не меньше другого). Если специальный критерий сортировки не

¹ В исходной версии STL отображение определялось в заголовочном файле `<map.h>`, а мульти множество — в файле `<multimap.h>`.

указан, по умолчанию используется критерий `less`. Объект функции `less` сортирует элементы, сравнивая их оператором `<` (дополнительная информация об объекте `less` приводится на с. 306)¹.

Необязательный четвертый параметр шаблона определяет модель памяти (см. главу 15). По умолчанию используется модель `allocator`, определенная в стандартной библиотеке C++².

Возможности отображений и мультиотображений

Отображения и мультиотображения, как и другие ассоциативные контейнеры, обычно реализуются в виде сбалансированных бинарных деревьев (рис. 6.9). В стандарте такая реализация не оговорена, но она следует из требований к сложности операций над отображениями и мультиотображениями. В сущности, множества, мультимножества, отображения и мультиотображения используют один и тот же внутренний тип данных. Множества и мультимножества можно рассматривать как частные случаи соответственно отображений и мультиотображений, у которых значения тождественны ключам элементов. Следовательно, отображения и мультиотображения обладают всеми свойствами и поддерживают все операции множеств и мультимножеств. Впрочем, второстепенные различия все же существуют. Во-первых, элементы представляют собой пары «ключ/значение». Во-вторых, отображения могут использоваться в качестве ассоциативных массивов.

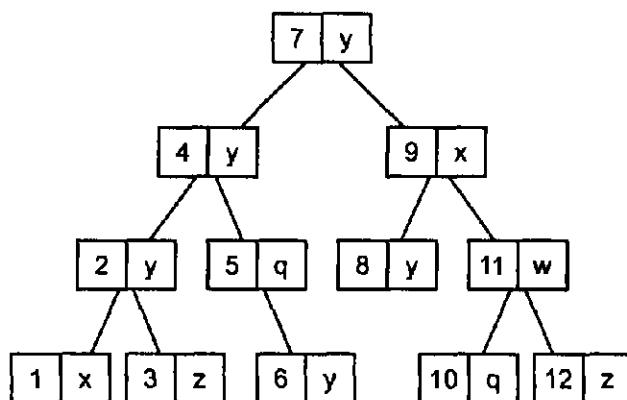


Рис. 6.9. Внутренняя структура отображений и мультиотображений

В отображениях и мультиотображениях элементы автоматически сортируются по ключу, поэтому эти контейнеры обеспечивают хорошее быстродействие при поиске элементов по известному ключу — операции, которая обычно выполняется относительно медленно. Автоматическая сортировка устанавливает важное ограничение для отображений и мультиотображений: ключ элемента нельзя изменить напрямую, потому что это нарушит упорядоченность элементов. Чтобы изменить ключ элемента, необходимо удалить элемент со старым

¹ В системах, не поддерживающих значения по умолчанию для параметров шаблонов, третий аргумент обычно является обязательным.

² В системах, не поддерживающих значения по умолчанию для параметров шаблонов, четвертый аргумент обычно отсутствует.

ключом, а затем вставить новый элемент с новым ключом и старым значением (подробности см. на с. 208). Как следствие, с точки зрения итератора ключ элемента является константным, однако прямая модификация *значения* элемента разрешена (если значение не принадлежит к константному типу).

Операции над отображениями и мультиотображениями

Операции создания, копирования и уничтожения

В табл. 6.26 представлены конструкторы и деструктор отображений и мультиотображений.

Таблица 6.26. Конструкторы и деструктор отображений и мультиотображений

Операция	Описание
map c	Создает пустое отображение или мультиотображение, не содержащее ни одного элемента
map c(op)	Создает пустое отображение или мультиотображение, использующее критерий сортировки op
map c1(c2)	Создает копию другого отображения или мультиотображения того же типа (с копированием всех элементов)
map c(beg,end)	Создает отображение или мультиотображение, инициализированное элементами интервала [beg,end)
map c(beg,end,op)	Создает отображение или мультиотображение с критерием сортировки op, инициализированное элементами интервала [beg,end)
c. \sim map()	Уничтожает все элементы и освобождает память

В таблице слово «set» обозначает собой одну из следующих конструкций:

- map<Elem> — отображение с сортировкой ключей по критерию less<> (оператор <);
- map<Elem,op> — отображение с сортировкой ключей по критерию op;
- multimap<Elem> — мультиотображение с сортировкой ключей по критерию less<> (оператор <);
- multimap<Elem,op> — мультиотображение с сортировкой ключей по критерию op.

Существуют два варианта определения критерия сортировки.

- В параметре шаблона, например¹:

```
std::map<float,std::string>; std::greater<float> > coll;
```

В этом случае критерий сортировки является частью типа. Таким образом, система типов гарантирует, что объединение возможно только для контейне-

¹ Обратите внимание на пробел между символами >. Последовательность > воспринимается компилятором как оператор сдвига, что приводит к синтаксической ошибке.

ров с одним критерием сортировки. Этот способ определения критерия сортировки является наиболее распространенным. Выражаясь точнее, в третьем параметре передается *тип* критерия сортировки, а конкретный критерий — это объект функции, создаваемый в контейнере. Для этого конструктор контейнера вызывает конструктор по умолчанию типа критерия сортировки. Пример с определением пользовательского критерия сортировки приведен на с. 296.

- В параметре конструктора. В этом варианте можно определить тип для нескольких критериев сортировки с разными значениями или состояниями этих критериев. Такой подход удобен при формировании критериев сортировки на стадии выполнения, а также при использовании различающихся критериев сортировки, которые относятся к одному типу данных. Пример приведен на с. 218.

Если критерий сортировки не указан, по умолчанию используется объект функции `less<>`, сортирующий элементы оператором `<!`.

Полное имя типа для контейнера получается излишне сложным и громоздким, поэтому для него рекомендуется определить псевдоним (то же самое полезно сделать и для определений итераторов):

```
typedef std::map<std::string, float, std::greater<string> >
    StringFloatMap;
...
StringFloatMap coll;
```

Конструктор, которому передается начало и конец интервала, может применяться для инициализации контейнера элементами контейнеров, относящихся к другим типам (от массива до стандартного входного потока данных). За подробностями обращайтесь к с. 153. Однако в данном случае элементы представляют собой пары «ключ/значение», поэтому необходимо проследить за тем, чтобы элементы исходного интервала относились к типу `pair<ключ, значение>` или могли быть преобразованы к нему.

Немодифицирующие операции над отображениями и мультиотображениями

Отображения и мультиотображения поддерживают обычный набор операций для получения размера контейнера и сравнения элементов (табл. 6.27).

Таблица 6.27. Немодифицирующие операции над отображениями и мультиотображениями

Операция	Описание
<code>c.size()</code>	Возвращает фактическое количество элементов
<code>c.empty()</code>	Проверяет, пуст ли контейнер (эквивалент <code>size() == 0</code> , но иногда выполняется быстрее)

¹ В системах, не поддерживающих значения по умолчанию для параметров шаблонов, критерий сортировки обычно является обязательным параметром:

`set<int, less<int> > coll;`

Операция	Описание
<code>c.max_size()</code>	Возвращает максимально возможное количество элементов
<code>c1 == c2</code>	Проверяет равенство <code>c1</code> и <code>c2</code>
<code>c1 != c2</code>	Проверяет неравенство <code>c1</code> и <code>c2</code> (эквивалент <code>!(c1==c2)</code>)
<code>c1 < c2</code>	Проверяет, что <code>c1</code> меньше <code>c2</code>
<code>c1 > c2</code>	Проверяет, что <code>c1</code> больше <code>c2</code> (эквивалент <code>c2 < c1</code>)
<code>c1 <= c2</code>	Проверяет, что <code>c1</code> не больше <code>c2</code> (эквивалент <code>!(c2 < c1)</code>)
<code>c1 >= c2</code>	Проверяет, что <code>c1</code> не меньше <code>c2</code> (эквивалент <code>!(c1 < c2)</code>)

Операции сравнения определены только для однотипных контейнеров. Это означает совпадение ключа, значения и критерия сортировки; в противном случае происходит ошибка компиляции. Пример:

```
std::map<float, std::string> c1; // Критерий сортировки: less<>
std::map<float, std::string, std::greater<float> > c2;
...
if (c1 == c2) {      // ОШИБКА: разные типы
    ...
}
```

Отношение «меньше/больше» между контейнерами проверяется по лексикографическому критерию (см. с. 356). Для сравнения контейнеров разных типов (с разными критериями сортировки) необходимо использовать алгоритмы, описанные на с. 352.

Специальные операции поиска

По аналогии с множествами и мультимножествами в отображениях и мультиотображениях определены специальные функции поиска, оптимизированные с учетом внутренней древовидной структуры контейнера (табл. 6.28).

Таблица 6.28. Специальные операции поиска в отображениях и мультиотображениях

Операция	Описание
<code>count(key)</code>	Возвращает количество элементов с ключом <code>key</code>
<code>find(key)</code>	Возвращает позицию первого элемента с ключом <code>key</code> (или <code>end()</code>)
<code>lower_bound(key)</code>	Возвращает первую позицию, в которой может быть вставлен элемент с ключом <code>key</code> (первый элемент с ключом <code>>= key</code>)
<code>upper_bound(key)</code>	Возвращает последнюю позицию, в которой может быть вставлен элемент с ключом <code>key</code> (первый элемент с ключом <code>> key</code>)
<code>equal_range(key)</code>	Возвращает первую и последнюю позиции, в которых может быть вставлен элемент с ключом <code>key</code> (интервал, в котором ключи равны <code>key</code>)

Функция `find()` ищет первый элемент с заданным ключом и возвращает его позицию в виде итератора. Если поиск оказывается безуспешным, функция

`find()` возвращает `end()`. Функция `find()` не может использоваться для поиска элемента с заданным *значением*. Вместо этого приходится задействовать универсальный алгоритм типа `find_if()` или программировать цикл. Ниже приведен пример простого цикла, выполняющего некоторую операцию для каждого элемента с заданным значением:

```
std::multimap<std::string,float> coll;
...
// Выполнение операции над всеми элементами с заданным значением
std::multimap<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    if (pos->second == value) {
        do_something();
    }
}
```

Если цикл используется для удаления элементов, будьте осторожны — по ошибке можно «отпилить ветку, на которой сидишь». За подробностями обратитесь на с. 211.

Искать элемент с заданным значением при помощи алгоритма `find_if()` еще сложнее, чем запрограммировать цикл, поскольку для этого придется создать объект функции, сравнивающий значение элемента с заданным значением. Пример приведен на с. 217.

Функции `lower_bound()`, `upper_bound()` и `equal_range()` работают так же, как во множествах (см. с. 188), за исключением того, что позиция определяется по ключу.

Присваивание

В отображениях и мультиотображениях определены только базовые операции присваивания, поддерживаемые всеми контейнерами (табл. 6.29). Дополнительная информация приведена на с. 156.

Таблица 6.29. Операции присваивания для отображений и мультиотображений

Операция	Описание
<code>c1 = c2</code>	Присваивает <code>c1</code> все элементы <code>c2</code>
<code>c1.swap(c2)</code>	Меняет местами содержимое <code>c1</code> и <code>c2</code>
<code>swap(c1,c2)</code>	То же, но в форме глобальной функции

Для выполнения операций присваивания контейнеры должны относиться к одному типу. В частности, должны совпадать типы критериев сравнения, хотя сами критерии могут различаться. Пример использования разных критериев сортировки, относящихся к одному типу, приведен на с. 198. Если критерии различаются, в результате присваивания или перестановки они также меняются местами.

Функции получения итераторов

Отображения и мультиотображения не поддерживают прямой доступ к элементам, поэтому для обращения к элементам обычно используются итераторы.

Впрочем, у этого правила существует исключение: отображения поддерживают оператор индексирования [] для прямого обращения к элементам (см. с. 212). В табл. 6.30 перечислены стандартные функции получения итераторов, поддерживаемые отображениями и мультиотображениями.

Таблица 6.30. Операции получения итераторов для отображений и мультиотображений

Операция	Описание
c.begin()	Возвращает двунаправленный итератор для первого элемента (ключи считаются константными)
c.end()	Возвращает двунаправленный итератор для позиции за последним элементом (ключи считаются константными)
c.rbegin()	Возвращает обратный итератор для первого элемента при переборе в обратном направлении
c.rend()	Возвращает обратный итератор для позиции за последним элементом при переборе в обратном направлении

Итераторы отображений и мультиотображений, как и во всех классах ассоциативных контейнеров, являются двунаправленными (см. с. 261). Такие итераторы не могут использоваться алгоритмами, рассчитанными на итераторы произвольного доступа (например, алгоритмами сортировки или случайной перестановки элементов).

Но еще более важное ограничение заключается в том, что с точки зрения итератора все ключи элементов отображения или мультиотображения считаются константными (то есть элемент интерпретируется как относящийся к типу `pair<const key,T>`). Это необходимо для того, чтобы программа не могла нарушить упорядоченность элементов, изменяя их ключи. Однако в результате для элементов отображения или мультиотображения вызов модифицирующих алгоритмов становится невозможным. Например, удаление элементов не может осуществляться алгоритмом `remove()`, потому что «удаление» в действительности сводится к перезаписи следующими элементами (данная тема подробно обсуждается на с. 125). Элементы множеств и мульти множеств удаляются только функциями, предоставляемыми контейнером.

Пример использования итераторов:

```
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;
}
```

Итератор `pos` используется для перебора в последовательности пар `string/float`. Выражение `pos->first` определяет ключ элемента, а выражение `pos->second` — значение элемента¹.

¹ `pos->first` является сокращенной записью для `(*pos).first`. Некоторые старые библиотеки поддерживают только полную форму записи.

Попытка изменения ключа приводит к ошибке:

```
pos->first = "hello"; // ОШИБКА компиляции
```

Однако модификация значения элемента выполняется без проблем (при условии, что значение не принадлежит к константному типу):

```
pos->second = 13.5; // OK
```

Изменить ключ элемента можно только одним способом: заменить старый элемент новым элементом с тем же значением. Унифицированная функция для выполнения этой операции выглядит так:

```
// cont/newkey.hpp
namespace MyLib {
    template <class 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()) {
            // Вставка нового элемента со значением старого элемента
            c.insert(typename Cont::value_type(new_key,
                                                pos->second));
            // Удаление старого элемента
            c.erase(pos);
            return true;
        }
        else {
            // Ключ не найден
            return false;
        }
    }
}
```

Функции `insert()` и `erase()` описаны в следующем подразделе.

При вызове этой унифицированной функции контейнеру просто передаются два ключа: старый и новый. Пример:

```
std::map<std::string, float> coll;
...
MyLib::replace_key(coll, "old key", "new key");
```

С мультиотображениями функция работает аналогично.

Впрочем, отображения поддерживают более удобный способ модификации ключа элемента. Вместо вызова `replace_key()` достаточно написать следующее:

```
// Вставка нового элемента со значением старого элемента
coll["new_key"] = coll["old_key"];
```

```
// Удаление старого элемента
coll.erase("old_key");
```

Использование оператора индексирования с отображениями рассматривается на с. 212.

Вставка и удаление элементов

В табл. 6.31 перечислены операции вставки и удаления элементов в отображениях и мультиотображениях.

Таблица 6.31. Операции вставки и удаления для отображений и мультиотображений

Операция	Описание
c.insert(elem)	Вставляет копию elem и возвращает позицию нового элемента; для отображений также возвращается признак успешного выполнения операции
c.insert(pos,elem)	Вставляет копию elem и возвращает позицию нового элемента (pos определяет рекомендуемую позицию, с которой следует начинать поиск позиции вставляемого элемента)
c.insert(beg,end)	Вставляет копию всех элементов интервала [beg,end) (и не возвращает значения)
c.erase(elem)	Удаляет все элементы со значением elem и возвращает количество удаленных элементов
c.erase(pos)	Удаляет элемент в позиции итератора pos (не возвращает значения)
c.erase(beg,end)	Удаляет все элементы из интервала [beg,end) (не возвращает значения)
c.clear()	Удаляет все элементы (контейнер остается пустым)

Замечания на с. 191 относительно операций вставки и удаления для миожеств и мультимиожеств справедливы и для отображений с мультиотображениями. В частности, тип возвращаемого значения этих функций тоже зависит от типа контейнера. Но использование функций несколько усложняется тем, что элемент представляет собой пару «ключ/значение».

При вставке пары «ключ/значение» необходимо учитывать, что внутри отображений и мультиотображений ключ считается константным. Вы должны либо предоставить правильный тип, либо обеспечить явное или неявное преобразование типа. Существуют три способа передачи значения при вставке нового элемента.

- *Использование обозначения value_type.* Для предотвращения неявного преобразования типа правильный тип указывается явно с использованием обозначения `value_type`, предоставляемого контейнером в виде определения типа.

Пример:

```
std::map<std::string, float> coll;
...
coll.insert(std::map<std::string, float>::value_type("otto",
22.3));
```

- *Использование типа pair<>*. Другой способ основан на непосредственном использовании типа `pair`. Пример:

```
std::map<std::string, float> coll;
...
// С неявным преобразованием
coll.insert(std::pair<std::string, float>("otto", 22.3));
// С явным преобразованием
coll.insert(std::pair<const std::string, float>("otto", 22.3));
```

В первой команде `insert()` тип указан неточно, поэтому он приводится к реальному типу элементов. Для этого функция `insert()` должна быть определена как шаблонная функция класса¹.

- *Использование функции make_pair()*. Вероятно, самый удобный способ основан на использовании функции `make_pair()` (см. с. 53). Функция создает объект `pair` из двух компонентов, переданных в аргументах:

```
std::map<std::string, float> coll;
...
coll.insert(std::make_pair("otto", 22.3));
```

В этом случае необходимые преобразования типа также выполняются шаблонной функцией `insert()`.

В следующем простом примере вставки элемента в отображение мы также проверяем, успешна ли завершилась операция:

```
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;
}
```

Возвращаемые значения функции `insert()` описаны на с. 191; там же приведены примеры использования этой функции, применимые и к отображениям. Стоит снова напомнить, что в отображениях предусмотрен более удобный способ вставки (и присваивания) элементов с помощью оператора индексирования (см. с. 212).

Чтобы удалить элемент с известным ключом, достаточно вызвать функцию `erase()`:

```
std::map<std::string, float> coll;
...
// Удаление всех элементов с заданным ключом
coll.erase(key);
```

¹ Если ваша система не поддерживает шаблонные функции классов, передайте элемент с правильным типом. Обычно для этого необходимо произвести явное преобразование типа.

Эта версия `erase()` возвращает количество удаленных элементов. Для отображений возвращается либо 0, либо 1.

Если мультиотображение содержит дубликаты, вам не удастся использовать функцию `erase()` для удаления только первого дубликата. Вместо этого можно воспользоваться следующим фрагментом:

```
typedef std::multimap<std::string, float> StringFloatMMap;
StringFloatMMap coll;
...
// Удаление первого элемента с переданным ключом
StringFloatMMap::iterator pos;
pos = coll.find(key);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

Вместо алгоритма `find()` используется функция класса `find()`, потому что она работает быстрее (см. пример на с. 163). Функция `find()` не годится для удаления элементов по известному значению (вместо ключа). Эта тема подробно обсуждается на с. 205.

При удалении элементов необходима осторожность, иначе вы рискуете «отпилить ветку, на которой сидите», то есть удалить элемент, на который ссылается итератор. Пример:

```
typedef std::multimap<std::string, float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    if (pos->second == value) {
        coll.erase(pos); // ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ!!!
    }
}
```

После вызова `erase()` для элемента, на который ссылается итератор `pos`, итератор становится недействительным. Любые попытки использования `pos` после удаления элемента без повторной инициализации — даже команда `++pos` — приводят к непредсказуемым последствиям.

Если бы функция `erase()` всегда возвращала значение следующего элемента, проблема решалась бы просто:

```
typedef std::multimap<std::string, float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
for (pos = coll.begin(); pos != coll.end(); ) {
    if (pos->second == value) {
```

```

    pos = coll.erase(pos);           // Хорошо бы, но...
}                                // происходит ОШИБКА КОМПИЛЯЦИИ!
else {
    ++pos;
}
}
}

```

Проектировщики решили не возвращать следующее значение, потому что если эта возможность не используется в программе, она просто означает лишние затраты времени. Автор не согласен с подобным решением, поскольку оно способствует появлению ошибок и усложнению программ (а в конечном счете может обернуться еще большими затратами времени).

Правильный способ удаления элемента, на который ссылается итератор, выглядит так:

```

typedef std::multimap<std::string, float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;

// Удаление всех элементов с заданным значением
for (pos = coll.begin(); pos != coll.end(); ) {
    if (pos->second == value) {
        coll.erase(pos++);
    }
    else {
        ++pos;
    }
}

```

Постфиксная команда `pos++` переводит итератор `pos` к следующему элементу, но возвращает копию исходного значения. Следовательно, в момент вызова `erase()` итератор `pos` не ссылается на удаляемый элемент.

Отображения как ассоциативные массивы

Ассоциативные контейнеры обычно не предоставляют прямого доступа к своим элементам; все обращения к элементам производятся через итераторы. Впрочем, отображения являются исключением из этого правила. Неконстантные отображения поддерживают оператор индексирования `[]` для прямого доступа к элементам (табл. 6.32). Тем не менее в качестве «индекса» используется не целочисленная позиция элемента, а ключ, предназначенный для его идентификации. Это означает, что индекс может относиться к произвольному типу. Подобный интерфейс характерен для так называемых *ассоциативных массивов*.

Таблица 6.32. Прямой доступ к элементам отображения оператором []

Операция	Описание
<code>m[key]</code>	Возвращает ссылку на значение элемента с ключом <code>key</code> . Вставляет элемент с ключом <code>key</code> , если его не существует

Ассоциативные массивы отличаются от обычных не только типом индекса. Индекс ассоциативного массива в принципе не может быть неправильным. Если элемента с заданным ключом не существует, в отображение автоматически вставляется новый элемент, значение которого инициализируется конструктором по умолчанию соответствующего типа. Чтобы эта схема работала, тип значения обязательно должен иметь конструктор по умолчанию. Базовые типы данных предоставляют конструктор по умолчанию, который инициализирует их значения нулями (см. с. 30).

У ассоциативных массивов есть как свои достоинства, так и недостатки.

- Основное достоинство – возможность использования более удобного интерфейса при вставке новых элементов в отображение. Пример:

```
std::map<std::string, float> coll; // Пустая коллекция
```

```
/* Вставка пары "otto"/7.7
 * - сначала вставляется пара "otto"/float()
 * - затем присваивается 7.7
 */
coll["otto"] = 7.7;
```

Обратите внимание на следующую команду:

```
coll["otto"] = 7.7;
```

Она работает следующим образом.

1) Обработка выражения `coll["otto"]`.

Если элемент с ключом "otto" существует, выражение возвращает значение элемента по ссылке. Если элемента с ключом "otto" не существует (как в нашем примере), в контейнер автоматически вставляется новый элемент с ключом "otto", причем значение элемента определяется конструктором по умолчанию для типа значения, и далее возвращается ссылка на значение нового элемента.

2) Присваивание числа 7.7 значению нового или существующего элемента.

В результате в отображении появляется элемент с ключом "otto" и значением 7.7.

- К недостаткам ассоциативных массивов относится возможность непреднамеренной вставки элементов. Скорее всего, результат выполнения следующей команды окажется неожиданным для программиста:

```
std::cout << coll["ottto"]
```

Команда вставляет в контейнер новый элемент с ключом "ottto" и выводит его значение, по умолчанию равное 0. На самом деле следовало бы вывести сообщение об ошибке с указанием на неверное написание "otto".

Также следует учитывать, что этот способ вставки медленнее обычного, описанного на с. 209. Дело в том, что новое значение сначала инициализируется значением по умолчанию для своего типа, которое затем заменяется правильным значением.

Обработка исключений

В отношении исключений отображения и мультиотображения предоставляют такие же гарантии, как множества и мультимножества (см. с. 194).

Примеры использования отображений и мультиотображений

Отображение как ассоциативный массив

В следующем примере показано, как использовать отображение с интерфейсом ассоциативного массива. В отображении хранятся данные биржевых котировок. Элемент представляет собой пару, в которой ключом является название акции, а значением — ее цена.

```
// cont/map1.cpp
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    /* Создание отображения / ассоциативного массива
     * - ключи - тип string
     * - значения - тип float
     */
    typedef map<string,float> StringFloatMap;

    StringFloatMap stocks;      // Создание пустого контейнера

    // Вставка нескольких элементов
    stocks["BASF"] = 369.50;
    stocks["VW"] = 413.50;
    stocks["Daimler"] = 819.00;
    stocks["BMW"] = 834.00;
    stocks["Siemens"] = 842.20;

    // Вывод всех элементов
    StringFloatMap::iterator pos;
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
        cout << "stock: " << pos->first << "\t"
            << "price: " << pos->second << endl;
    }
    cout << endl;

    // Биржевой бум (все цены удваиваются)
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
        pos->second *= 2;
    }
```

```
// Вывод всех элементов
for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock: " << pos->first << "\t"
        << "price: " << pos->second << endl;
}
cout << endl;

/* Переименование ключа "VW" в "Volkswagen"
 * - возможно только с созданием нового элемента
 */
stocks["Volkswagen"] = stocks["VW"];
stocks.erase("VW");

// Вывод всех элементов
for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock: " << pos->first << "\t"
        << "price: " << pos->second << endl;
}
}
```

Программа выводит следующий результат:

```
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
```

Мультиотображение как словарь

В следующем примере демонстрируется словарь (мультиотображение, в котором ключи и значения относятся к строковому типу).

```
// cont/mmap1.cpp
#include <iostream>
#include <map>
#include <string>
#include <iomanip>
using namespace std;
```

```
int main()
{
    // Определение словаря (мультимножества "string/string")
    typedef multimap<string,string> StrStrMMap;

    // Создание пустого словаря
    StrStrMMap dict;

    // Вставка нескольких элементов в произвольном порядке
    dict.insert(make_pair("day","Tag"));
    dict.insert(make_pair("strange","fremd"));
    dict.insert(make_pair("car","Auto"));
    dict.insert(make_pair("smart","elegant"));
    dict.insert(make_pair("trait","Merkmäl"));
    dict.insert(make_pair("strange","seltsam"));
    dict.insert(make_pair("smart","raffiniert"));
    dict.insert(make_pair("smart","klug"));
    dict.insert(make_pair("clever","raffiniert"));

    // Вывод всех элементов
    StrStrMMap::iterator pos;
    cout.setf (ios::left, ios::adjustfield);
    cout << ' ' << setw(10) << "english "
        << "german " << endl;
    cout << setfill('-') << setw(20) << ""
        << setfill(' ') << endl;
    for (pos = dict.begin(); pos != dict.end(); ++pos) {
        cout << ' ' << setw(10) << pos->first.c_str()
            << pos->second << endl;
    }
    cout << endl;

    // Вывод всех значений для ключа "smart"
    string word("smart");
    cout << word << ":" << endl;
    for (pos = dict.lower_bound(word);
         pos != dict.upper_bound(word); ++pos) {
        cout << "    " << pos->second << endl;
    }

    // Вывод всех ключей для значения "raffiniert"
    word = ("raffiniert");
    cout << word << ":" << endl;
    for (pos = dict.begin(); pos != dict.end(); ++pos) {
        if (pos->second == word) {
            cout << "    " << pos->first << endl;
        }
    }
}
```

Результат выполнения программы выглядит так:

```
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
```

Поиск элементов с известным значением

В следующем примере показано, как при помощи глобального алгоритма `find_if()` найти элемент с заданным значением.

```
// cont/mapfind.cpp
#include <iostream>
#include <algorithm>
#include <map>
using namespace std;

/* Объект функции, проверяющий значение элемента отображения
 */
template <class K, class V>
class value_equals {
private:
    V value;
public:
    // Конструктор (инициализация значения, используемого при сравнении)
    value_equals (const V& v)
        : value(v) {
    }
    // Сравнение
    bool operator() (pair<const K, V> elem) {
        return elem.second == value;
    }
};

int main()
```

```

{
    typedef map<float,float> FloatFloatMap;
    FloatFloatMap coll;
    FloatFloatMap::iterator pos;

    // Заполнение контейнера
    coll[1]=7;
    coll[2]=4;
    coll[3]=2;
    coll[4]=3;
    coll[5]=6;
    coll[6]=1;
    coll[7]=3;

    // Поиск элемента с ключом 3.0
    pos = coll.find(3.0);           // Логарифмическая сложность
    if (pos != coll.end()) {
        cout << pos->first << ":" <<
            pos->second << endl;
    }

    // Поиск элемента со значением 3.0
    pos = find_if(coll.begin(),coll.end(), // Линейная сложность
                  value_equal<float,float>(3.0));
    if (pos != coll.end()) {
        cout << pos->first << ":" <<
            pos->second << endl;
    }
}

```

Результат выполнения программы выглядит так:

```

3: 2
4: 3

```

Пример с отображениями, строками и изменением критерия сортировки на стадии выполнения

Следующий пример предназначен для опытных программистов, хорошо разбирающихся в STL. Он дает представление как о мощи STL, так и о некоторых недостатках, присущих этой библиотеке. В частности, демонстрируются приемы:

- использования отображений;
- создания и использования объектов функций;
- определения критерия сортировки во время выполнения программы;
- сравнения строк без учета регистра символов.

```

// cont/mapcmp.cpp
#include <iostream>
#include <iomanip>

```

```
#include <map>
#include <string>
#include <algorithm>
using namespace std;

/* Объект функции для сравнения строк
 * - позволяет задать критерий сравнения во время выполнения
 * - позволяет сравнивать символы без учета регистра
 */
class RuntimeStringCmp {
public:
    // Константы режима сравнения
    enum cmp_mode {normal, nocase};
private:
    // Используемый 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: инициализация критерия 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);
        }
    }
};

/* Тип контейнера:
 * - отображение
 *     - ключ: string
 *     - значение: string
 *     - специальный тип критерия сравнения
 */
typedef map<string, string, RuntimeStringCmp> StringStringMap;

// Функция для заполнения контейнера и вывода его содержимого
void fillAndPrint(StringStringMap& coll);
```

```
int main()
{
    // Создание контейнера с критерием сравнения по умолчанию
    StringStringMap coll1;
    fillAndPrint(coll1);

    // Создание объекта для сравнений без учета регистра символов
    RuntimeStringCmp ignorecase(RuntimeStringCmp::nocase);

    // Создание контейнера с критерием сравнения без учета регистра
    StringStringMap coll2(ignorecase);
    fillAndPrint(coll2);
}

void fillAndPrint(StringStringMap& coll)
{
    // Вставка элементов в произвольном 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
    StringStringMap::iterator pos;
    cout.setf(ios::left, ios::adjustfield);
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        cout << setw(15) << pos->first.c_str() << " "
            << pos->second << endl;
    }
    cout << endl;
}
```

Функция `main()` создает два контейнера и вызывает для них функцию `fillAndPrint()`, которая заполняет контейнеры одинаковыми элементами и выводит их содержимое. Однако контейнеры используют разные критерии сортировки.

- Контейнер `coll1` использует объект функции типа `RuntimeStringCmp`, по умолчанию сравнивающий элементы оператором `<`.
- Контейнер `coll2` использует объект функции типа `RuntimeStringCmp`, который инициализируется значением `nocase`, определенным в классе `RuntimeStringCmp`. При включении режима `nocase` объект функции сортирует строки без учета регистра символов.

Результат выполнения программы выглядит так:

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

В первой части итоговых данных выводится содержимое первого контейнера, сравнивающего элементы оператором <. Сначала перечисляются все ключи, начинающиеся с символов верхнего регистра, а за ними следуют ключи, начинающиеся с символов нижнего регистра.

Во второй части ключи сравниваются без учета регистра, поэтому порядок перечисления элементов изменяется. Обратите внимание — вторая часть содержит на одну строку меньше первой. Дело в том, что при сравнении без учета регистра слова «Unternehmen» и «unternehmen»¹ оказываются равными, а критерий сортировки нашего отображения не допускает наличия дубликатов. К сожалению, в результате возникает путаница — ключу, которому должно соответствовать значение «enterprise», соответствует значение «undertake». Вероятно, в этом примере было бы правильнее использовать мультиотображение. Обычно в качестве словарей применяется именно этот тип контейнера.

Другие контейнеры STL

Библиотека STL — это архитектура, а не набор классов. Кроме стандартных контейнерных классов она позволяет работать с другими структурами данных. Вместо контейнеров STL программист может использовать строки или обычные массивы или же написать собственный контейнер для особых целей. При этом он по-прежнему может применять алгоритмы (например, сортировки или слияния) для своего типа контейнера. Подобная архитектура является хорошим

¹ В немецком языке существительные начинаются с прописной, а глаголы — со строчной буквы.

примером «принципа открытой закрытости»¹: библиотека *открыта* для расширения, но *закрыта* для модификации.

Существуют три подхода к созданию «STL-совместимых» контейнеров:

- *Активный подход*. Программист реализует интерфейс, обязательный для контейнеров STL. В частности, он должен реализовать обычные функции контейнеров вроде `begin()` и `end()`. Такой подход называется активным, поскольку требует, чтобы контейнер был написан по определенным правилам.
- *Пассивный подход*. Программист создает или использует специальные итераторы, обеспечивающие интерфейс между алгоритмами и специализированными контейнерами. Все, чего требует пассивный подход, — возможности перебора элементов контейнера, которая в том или ином виде поддерживается любым контейнером.
- *Интерфейсные классы*. Программист объединяет два предыдущих подхода и пишет интерфейсный класс, который инкапсулирует реальную структуру данных в интерфейсе, отвечающем требованиям к контейнерам STL.

В этом разделе сначала рассматривается использование строк как стандартных контейнеров (пример активного подхода). Затем пассивный подход будет описан на примере другого контейнера — обычного массива. Впрочем, для работы с данными обычного массива также подходят интерфейсные классы. В конце раздела обсуждаются некоторые аспекты, относящиеся к хэш-таблице, — важному контейнеру, не описанному в стандарте.

При написании собственного контейнера STL также можно обеспечить поддержку его параметризации для разных распределителей памяти. Стандартная библиотека C++ содержит набор специальных функций и классов для программирования с распределителями и неинициализированной памятью. За дополнительной информацией обращайтесь на с. 702.

Строки как контейнеры STL

Строковые классы стандартной библиотеки C++ являются собой пример активного подхода к написанию контейнеров STL (строковые классы рассматриваются в главе 11). Строку можно рассматривать как контейнер для хранения символов. Символы строки образуют последовательность; перебирая элементы этой последовательности, можно получить доступ к отдельным символам. Таким образом, стандартные строковые классы поддерживают интерфейс контейнеров STL. В них определены функции `begin()` и `end()`, возвращающие итераторы прямого доступа для перебора символов строки, и функции для итераторов и итераторных адаптеров (например, функция `push_back()` для вставки элементов с конца).

Учтите, что работа со строками в контексте STL выглядит несколько необычно. Дело в том, что обычно строка обрабатывается как единое целое (при передаче, копировании и присваивании используются целые строки, а не составляющие их символы). Но при работе со строкой на уровне отдельных символов (напри-

¹ Автор впервые услышал о «принципе открытой закрытости» от Роберта К. Мартина (Robert C. Martin), который, в свою очередь, узнал о нем от Бертрана Мейера (Bertrand Meyer).

мер, при чтении символов потоковым итератором ввода или преобразовании строки к верхнему или нижнему регистру) алгоритмы STL могут оказаться полезными. Вдобавок алгоритмы STL позволяют определять специальные критерии сравнения строк, тогда как стандартный интерфейс строк такой возможности не предоставляет.

Различные аспекты использования строк в STL с примерами будут рассмотрены в главе, посвященной строкам (см. с. 480).

Обычные массивы как контейнеры STL

Обычные массивы могут использоваться как контейнеры STL. Однако обычный массив не является классом, поэтому он не содержит функций `begin()` и `end()` и для него невозможно определять функции. В таких случаях приходится использовать либо пассивный подход, либо интерфейсные классы.

Непосредственное использование обычного массива

Вариант с пассивным подходом реализуется просто. Вам понадобятся лишь объекты, обеспечивающие перебор элементов массива через интерфейс итераторов STL. И такие итераторы уже существуют — это обычные указатели. При проектировании STL было решено использовать для итераторов интерфейс указателей, потому что в этом случае обычные указатели превращаются в итераторы. Здесь в очередной раз проявляется общая концепция чистой абстракции: любой объект, который *ведет себя* как итератор, *является* итератором (см. с. 261). Пример использования обычного массива как контейнера STL:

```
// cont/array1.cpp
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{
    int coll[] = { 5, 6, 2, 4, 1, 3 };

    // Возведение элементов в квадрат
    transform (coll, coll+6,           // Первый источник
               coll,                 // Второй источник
               coll,                 // Приемник
               multiplies<int>()); // Операция

    // Сортировка выполняется, начиная со второго элемента
    sort (coll+1, coll+6);

    // Вывод всех элементов
    copy (coll, coll+6,
          ostream_iterator<int>(cout, " "));
    cout << endl;
}
```

Будьте внимательны и правильно передайте конец массива, как это сделано в нашем примере (`(coll+6)`). Как обычно, конец интервала находится в позиции *за последним элементом*.

Результат выполнения программы выглядит так:

```
25 1 4 9 16 36
```

Дополнительные примеры приведены на с. 372 и 411.

Интерфейсный класс для массива

В своей книге «The C++ Programming Language», 3rd Edition, Бъярн Страуструп описывает очень полезный интерфейсный класс для обычных массивов. Он гораздо надежнее обычных массивов, хотя и не уступает им по быстродействию. Наконец, он является хорошим примером контейнера STL, определяемого пользователем. В данном случае используется вариант с интерфейсным классом, то есть стандартный контейнерный интерфейс реализуется как оболочка вокруг массива.

Определение класса `carray` (сокращенное от «C array» или «constant size array») выглядит так¹:

```
// cont/carray.hpp
#include <cstddef>

template<class T, std::size_t thesize>
class carray {
private:
    T v[thesize];           // Массив фиксированного размера
                           // для элементов типа T
public:
    // Определения типов
    typedef T      value_type;
    typedef T*     iterator;
    typedef const T* const_iterator;
    typedef T&     reference;
    typedef const T& const_reference;
    typedef std::size_t   size_type;
    typedef std::ptrdiff_t difference_type;

    // Поддержка итераторов
    iterator begin() { return v; }
    const_iterator begin() const { return v; }
    iterator end() { return v+thesize; }
    const_iterator end() const { return v+thesize; }

    // Прямой доступ к элементам
    reference operator[](std::size_t i) { return v[i]; }
    const_reference operator[](std::size_t i) const { return v[i]; }
```

¹ Исходный интерфейсный класс Бъярна Страуструпа называется `c_array`, а его определение приведено в разделе 17.5.4 книги Страуструпа. В данном случае в интерфейсный класс внесены небольшие изменения.

```
// Фиксированный размер
size_type size() const { return thesize; }
size_type max_size() const { return thesize; }

// Преобразование к обычному массиву
T* as_array() { return v; }
}:
```

Пример использования класса `carray`:

```
// cont/carray1.cpp
#include <algorithm>
#include <functional>
#include "carray.hpp"
#include "print.hpp"
using namespace std;

int main()
{
    carray<int,10> a;

    for (unsigned i=0; i<a.size(); ++i) {
        a[i] = i+1;
    }
    PRINT_ELEMENTS(a);

    reverse(a.begin(),a.end());
    PRINT_ELEMENTS(a);

    transform(a.begin(),a.end(),
              a.begin(),
              negate<int>()); // Источник
                           // Приемник
                           // Операция
    PRINT_ELEMENTS(a);
}
```

Как видно из приведенного кода, прямое взаимодействие с контейнером осуществляется через операции общего контейнерного интерфейса (функции `begin()`, `end()` и оператор `[]`). Следовательно, в вашем распоряжении также оказываются различные операции, вызывающие функции `begin()` и `end()`, — такие, как алгоритмы и вспомогательная функция `PRINT_ELEMENTS()`, представленная на с. 128.

Результат выполнения программы выглядит так:

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1
```

Хэш-таблицы

В стандартную библиотеку C++ не вошла такая важная структура данных, как хэш-таблица. Предложения о включении хэш-таблиц в стандарт были, однако хэш-таблицы не входили в исходную версию STL, и комитет решил, что предло-

жение об их включении поступило слишком поздно. В какой-то момент приходится прекратить прием новых предложений и сосредоточиться на освоении того, что есть; иначе работа никогда бы не была закончена.

В сообществе C++ существует несколько распространенных реализаций C++. В библиотеках обычно определяются четыре разновидности хэш-таблиц: `hash_set`, `hash_multiset`, `hash_map` и `hash_multimap`. По аналогии со стандартными ассоциативными контейнерами `multi`-версии допускают присутствие дубликатов, а таблицы `hash_map` и `hash_multimap` содержат пары «ключ/значение». Баян Страуструп подробно рассматривает хэш-таблицы `hash_map` как пример вспомогательного контейнера STL в разделе 17.6 своей книги «The C++ Programming Language», 3rd Edition. Примеры конкретной реализации хэш-контейнеров можно найти, например, в версии STLport (<http://www.stlport.org/>). Учтите, что разные реализации могут расходиться в деталях, поскольку хэш-контейнеры еще не стандартизированы.

Реализация ссылочной семантики

Контейнерные классы STL поддерживают семантику значений, но не поддерживают ссылочную семантику. Они создают внутренние копии вставляемых элементов и затем возвращают эти копии. На с. 144 рассматриваются достоинства и недостатки обеих семантик, а также некоторые последствия, к которым приводит выбор. Напоминаем один из выводов: если вам потребуется ссылочная семантика в контейнерах STL (например, из-за того, что копирование элементов обходится слишком дорого или же элементы должны совместно использоваться несколькими коллекциями), воспользуйтесь классом умного указателя для предотвращения потенциальных ошибок. Ниже приведено одно из возможных решений проблемы. В нем задействован вспомогательный класс умного указателя с подсчетом ссылок на объекты, на которые ссылается указатель¹:

```
// cont/countptr.hpp
#ifndef COUNTED_PTR_HPP
#define COUNTED_PTR_HPP

/* Класс, обеспечивающий семантику подсчета ссылок
 * - объект, на который ссылается указатель, автоматически
 *   уничтожается при удалении последнего экземпляра CountedPtr
 *   для данного объекта.
 */
template <class T>
class CountedPtr {
private:
    T* ptr;           // Указатель на значение
    long* count;      // Количество владельцев (общие данные)
```

¹ Большое спасибо Грегу Колвину (Greg Colvin) и Биману Доуэсу (Biman Dowes) за помощь в реализации этого класса.

```
public:
    // Инициализация объекта существующим указателем
    // - указатель p должен быть получен в результате вызова new
    explicit CountedPtr (T* p=0)
        : ptr(p), count(new long(1)) {
    }

    // Копирующий указатель (увеличивает счетчик владельцев)
    CountedPtr (const CountedPtr<T>& p) throw()
        : ptr(p.ptr), count(p.count) {
            ++*count;
    }

    // Деструктор (уничтожает объект, если владелец был последним)
    ~CountedPtr () throw() {
        dispose();
    }

    // Присваивание (перевод указателя на новый объект)
    CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {
        if (this != &p) {
            dispose();
            ptr = p.ptr;
            count = p.count;
            ++*count;
        }
        return *this;
    }

    // Доступ к объекту, на который ссылается указатель
    T& operator*() const throw() {
        return *ptr;
    }
    T* operator->() const throw() {
        return ptr;
    }

private:
    void dispose() {
        if (--*count == 0) {
            delete count;
            delete ptr;
        }
    }
};

#endif /*COUNTED_PTR_HPP*/
```

Класс напоминает стандартный класс умного указателя `auto_ptr` (см. с. 54). Предполагается, что значения, которыми инициализируются умные указатели, были возвращены оператором `new`. В отличие от класса `auto_ptr` это позволяет копировать умные указатели так, чтобы оригинал и копия оставались действительными. Объект уничтожается только после удаления последнего указателя, ссылающегося на него.

Класс `CounterPtr` можно усовершенствовать, например реализовать в нем автоматическое преобразование типов или возможность передачи права владения от умного указателя вызывающей стороне.

Ниже приведен пример использования класса `CountedPtr`:

```
// cont/refsem1.cpp
#include <iostream>
#include <list>
#include <deque>
#include <algorithm>
#include "countptr.hpp"
using namespace std;

void printCountedPtr (CountedPtr<int> elem)
{
    cout << *elem << ' ';
}

int main()
{
    // Массив целых чисел (для совместного использования
    // в разных контейнерах)
    static int values[] = { 3, 5, 9, 1, 6, 4 };

    // Две разные коллекции
    typedef CountedPtr<int> IntPtr;
    deque<IntPtr> coll1;
    list<IntPtr> coll2;

    /* Вставка общих объектов в коллекции
     * - исходный порядок в coll1
     * - обратный порядок в coll2
     */
    for (int i=0; i<sizeof(values)/sizeof(values[0]); ++i) {
        IntPtr ptr(new int(values[i]));
        coll1.push_back(ptr);
        coll2.push_front(ptr);
    }

    // Вывод содержимого обеих коллекций
    for_each (coll1.begin(), coll1.end(),
              printCountedPtr);
    cout << endl;
```

```
for_each (coll2.begin(), coll2.end(),
          printCountedPtr);
cout << endl << endl;

/* Модификация значений в разных коллекциях
 * - возвведение в квадрат третьего значения в coll1
 * - изменение знака первого значения в coll1
 * - обнуление первого значения в coll2
 */
*coll1[2] *= *coll1[2];
(**coll1.begin()) *= -1;
(**coll2.begin()) = 0;

// Повторный вывод содержимого обеих коллекций
for_each (coll1.begin(), coll1.end(),
          printCountedPtr);
cout << endl;
for_each (coll2.begin(), coll2.end(),
          printCountedPtr);
cout << endl;
}
```

Результат выполнения программы выглядит так:

```
3 5 9 1 6 4
4 6 1 9 5 3

-3 5 81 1 6 0
0 6 1 81 5 -3
```

Если вызвать вспомогательную функцию, сохраняющую элемент коллекции (`IntPtr`) где-то в другом месте, то значение, на которое ссылается указатель, остается действительным даже после уничтожения коллекций или удаления из них всех элементов.

В архиве Boost библиотек C++ (<http://www.boost.org/>) хранятся различные классы умных указателей, расширяющие стандартную библиотеку C++ (вероятно, вместо `CounterPtr<>` стоит поискать название `shared_ptr<>`).

Рекомендации по выбору контейнера

Стандартная библиотека C++ содержит разные типы контейнеров, обладающие разными возможностями. Возникает вопрос: когда использовать тот или иной тип контейнера? В табл. 6.33 приводятся некоторые рекомендации. Однако эти общие утверждения не всегда согласуются с реальностью. Например, при малом количестве элементов фактор сложности можно не учитывать, потому что обработка малого количества элементов с линейной сложностью происходит быстрее, чем обработка большого количества элементов с логарифмической сложностью.

Таблица 6.33. Характеристики контейнеров STL

Вектор	Дек	Список	Множество	Мульти множество	Отображение	Мульти отображение
Типичная внутренняя реализация Элементы	Динамический массив Значение	Массив массивов Значение	Двусвязный список Значение	Бинарное дерево Значение	Бинарное дерево Пара «ключ/значение»	Бинарное дерево Пара «ключ/значение»
Возможность существования дубликатов	Да	Да	Нет	Да	Не для ключа	Да
Поддержка произвольного доступа	Да	Да	Нет	Нет	С ключом	Нет
Категория итератора	Произвольный доступ	Произвольный доступ	Двунаправленный	Двунаправленный (константные элементы)	Двунаправленный (константные элементы)	Двунаправленный (константные ключи)
Скорость поиска	Медленная	Медленная	Очень медленная	Быстрая	Быстрая для ключа	Быстрая для ключа
Быстрая вставка/удаление	В конце	В начале и в конце	Где угодно	—	—	—
Недействительность итераторов, ссылок и указателей при вставке/удалении	При перераспределении памяти	Всегда	Никогда	Никогда	Никогда	Никогда
Освобождение памяти от удаленных элементов	Никогда	Иногда	Всегда	Всегда	Всегда	Всегда
Возможность резервирования памяти	Да	Нет	—	—	—	—
Транзакционная безопасность (успешное выполнение или отсутствие изменений)	Присоединение/ удаление в конце	Присоединение/ удаление в начале и в конце	Все операции, кроме sort() и присваивания	Все операции, кроме вставки нескольких элементов	Все операции, кроме вставки нескольких элементов	Все операции, кроме вставки нескольких элементов

В дополнение к таблице далее приводится еще несколько полезных практических рекомендаций.

- По умолчанию выбирайте вектор. Этот контейнер имеет простейшую внутреннюю структуру и поддерживает произвольный доступ. Этим обусловлена гибкость и универсальность доступа, а обработка данных часто выполняется достаточно быстро.
- Если вы собираетесь часто вставлять и/или удалять элементы в начале и в конце последовательности, выбирайте дек. Кроме того, дек больше подходит в ситуациях, когда при удалении элементов обязательно должна освобождаться внутренняя память контейнера. Элементы вектора обычно хранятся в одном блоке памяти, поэтому дек может содержать больше элементов за счет использования нескольких блоков памяти.
- Если вы собираетесь часто вставлять, удалять и перемещать элементы в середине контейнера, возможно, вам стоит остановить выбор на списке. Списки поддерживают специальные функции для перемещения элементов между контейнерами с постоянным временем. Но учтите, что списки не обеспечивают произвольного доступа, поэтому обращение к элементам внутри списка выполняется относительно медленно (если известно только начало списка).
Как и во всех узловых контейнерах, итераторы списков остаются действительными до тех пор, пока элементы, на которые они ссылаются, остаются в контейнере. В векторах все итераторы, указатели и ссылки становятся недействительными при превышении емкости, а некоторые – при вставке и удалении элементов. Итераторы, указатели и ссылки в деках становятся недействительными при изменении размера контейнера.
- Если вам нужен контейнер с высоким уровнем безопасности исключений, когда любая операция либо завершается успешно, либо не вносит изменений, выбирайте список, но воздержитесь от выполнения присваивания и вызова функции `sort()`, а если исключения могут произойти при сравнении элементов – от вызова функций `merge()`, `remove()`, `remove_if()` и `unique()` (см. с. 180). Можно также выбрать ассоциативные контейнеры (но без многоэлементной вставки, а если исключения могут произойти при копировании/присваивании критерия сравнения – без вызова функции `swap()`). Общие сведения об обработке исключений в STL приведены на с. 148. На с. 254 перечислены все контейнерные операции, предоставляющие особые гарантии в отношении исключений.
- Если вам требуется часто искать элементы по определенному критерию, воспользуйтесь множеством или мультимножеством с сортировкой элементов по этому критерию. Помните, что при сортировке 1000 элементов логарифмическая сложность работает на порядок быстрее линейной. В таких ситуациях наглядно проявляются преимущества бинарных деревьев.

Скорость поиска по хэш-таблице обычно в 5–10 раз превышает скорость поиска по бинарному дереву. Подумайте об использовании хэш-контейнера, даже несмотря на то, что хэш-таблицы не стандартизированы. Однако содер-

жимое хэш-контейнеров не сортируется, и если вам необходима определенная упорядоченность элементов, этот тип контейнера вам не подойдет. А раз хэш-таблицы не входят в стандартную библиотеку C++, вам понадобятся их исходные тексты для обеспечения переносимости программы.

- Для работы с парами «ключ/значение» используется отображение или мультиотображение (или их хэшированные версии, если они доступны).
- Если вам нужен ассоциативный массив, используйте отображение.
- Если вам нужен словарь, используйте мультиотображение.

Если нужно сортировать объекты по двум разным критериям, возникают проблемы. Допустим, вы хотите хранить объекты в порядке, указанном пользователем, но при этом предоставить средства поиска по другому критерию; как и при работе с базами данных, операции по двум и более критериям должны выполняться достаточно быстро. Вероятно, в такой ситуации стоит воспользоваться двумя множествами или отображениями, совместно использующими общие объекты с разными критериями сортировки. С другой стороны, хранение объектов в двух коллекциях — особая тема, которая рассматривается на с. 226.

Автоматическая сортировка ассоциативных контейнеров вовсе не означает, что эти контейнеры работают более эффективно там, где сортировка необходима. Дело в том, что ассоциативный контейнер проводит сортировку при каждой вставке нового элемента. Часто более эффективным оказывается другое решение — использование последовательного контейнера и сортировка всех элементов после вставки одним или несколькими алгоритмами сортировки (см. с. 327).

Ниже приведены две простые программы, которые сортируют строки, прочитанные из стандартного входного потока данных, и выводят их без дубликатов. Задача решается с применением двух разных контейнеров.

Решение с использованием *множества*:

```
// cont/sortset.cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <set>
using namespace std;

int main()
{
    /* Создание строкового множества
     * - инициализация словами, прочитанными из стандартного ввода
     */
    set<string> coll((istream_iterator<string>(cin)),
                      (istream_iterator<string>()));

    // Вывод всех элементов
    copy (coll.begin(), coll.end(),
          ostream_iterator<string>(cout, "\n"));
}
```

Решение с использованием *вектора*:

```
// cont/sortvec.cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    /* Создание строкового вектора
     * - инициализация словами, прочитанными из стандартного ввода
     */
    vector<string> coll((istream_iterator<string>(cin)),
                         (istream_iterator<string>()));

    // Сортировка элементов
    sort (coll.begin(), coll.end());

    // Вывод всех элементов с подавлением дубликатов
    unique_copy (coll.begin(), coll.end(),
                  ostream_iterator<string>(cout, "\n"));
}
```

Когда автор запустил обе программы в своей системе для тестового набора из 150 000 строк, векторная версия работала примерно на 10 % быстрее. Включение вызова `reserve()` ускорило ее еще на 5 %. Если разрешить наличие дубликатов (использование типа `multiset` вместо `set` и вызов `copy()` вместо `unique_copy()`), ситуация кардинально меняется: векторная версия работает еще на 40 % быстрее! Такие показатели нельзя считать типичными, однако они доказывают, что во многих случаях стоит опробовать разные варианты обработки элементов.

На практике иногда бывает трудно предсказать, какой тип контейнера лучше подходит для конкретной задачи. Одно из больших достоинств STL заключается в том, что вы можете относительно легко опробовать разные варианты. Основная работа — реализация структур данных и алгоритмов — уже выполнена. Вам остается лишь скомбинировать ее результаты оптимальным образом.

Гипы и функции контейнеров

В настоящем разделе подробно описаны различные контейнеры STL и все поддерживаемые ими операции. Типы и операции сгруппированы по функциональности. Для каждого типа и операции приводятся сигнатура, краткое описание и типы контейнеров, в которых они поддерживаются. Под обозначением *контейнер* понимается тип контейнера (вектор, деск, список, множество, мультимножество, отображение, мультиотображение или строка).

Определения типов

`контейнер::value_type`

- Тип элементов.
- Для множеств и мульти множеств — константный тип.
- Для отображений и мультиотображений — тип:
`pair <const тип_ключа, тип_значения>`
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`контейнер::reference`

- Тип ссылки на элемент.
- В общем случае:
`контейнер::value_type&`
- Для `vector<bool>` — вспомогательный класс (см. с. 167).
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`контейнер::const_reference`

- Тип константной ссылки на элемент.
- В общем случае:
`const контейнер::value_type&`
- Для `vector<bool>` — тип `bool`.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`контейнер::iterator`

- Тип итератора.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`контейнер::const_iterator`

- Тип константного итератора.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`контейнер::reverse_iterator`

- Тип обратного итератора.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями.

`контейнер::const_reverse_iterator`

- Тип константного обратного итератора.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями.

`контейнер::size_type`

- Беззнаковый целый тип для значений размера.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`контейнер::difference_type`

- Знаковый целый тип для значений разности.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`контейнер::key_type`

- Тип ключа элементов в ассоциативных контейнерах.
- Поддерживается множествами, мульти множествами, отображениями, мультиотображениями.

`контейнер::mapped_type`

- Тип значения элементов ассоциативных контейнеров.
- Поддерживается отображениями и мультиотображениями.

`контейнер::key_compare`

- Тип критерия сравнения в ассоциативных контейнерах.
- Поддерживается множествами, мульти множествами, отображениями, мультиотображениями.

`контейнер::value_compare`

- Тип критерия сравнения для типа всего элемента.
- Для множеств и мульти множеств — эквивалент `key_compare`.
- Для отображений и мультиотображений — вспомогательный класс для критерия сравнения, при котором сравниваются только ключевые части двух элементов.
- Поддерживается множествами, мульти множествами, отображениями, мультиотображениями.

`контейнер::allocator_type`

- Тип распределителя.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

Операции создания, копирования и уничтожения

Ниже перечислены конструкторы и деструкторы контейнеров. Многие конструкторы позволяют в дополнительном аргументе передать также распределитель памяти (см. с. 234).

`контейнер::контейнер ()`

- Конструктор по умолчанию.
- Создает новый пустой контейнер.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`explicit контейнер::контейнер (const CompFunc& op)`

- Создает новый пустой контейнер с критерием сортировки *op* (см. с. 198 и 218).
- Критерий сортировки должен определять «строгую квазиупорядоченность» (см. с. 184).
- Поддерживается множествами, мульти множествами, отображениями, мультиотображениями.

`explicit контейнер::контейнер (const контейнер& c)`

- Копирующий конструктор.
- Создает новый контейнер как копию существующего контейнера *c*.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`explicit контейнер::контейнер (size_type num)`

- Создает контейнер с *num* элементами.
- Элементы создаются конструктором по умолчанию своего типа.
- Поддерживается векторами, деками, списками.

`explicit контейнер::контейнер (size_type num, const T& value)`

- Создает контейнер с *num* элементами.
- Элементы создаются как копии *value*.
- Элементы контейнера относятся к типу *T*.
- Для строк значение *value* не передается по ссылке.
- Поддерживается векторами, деками, списками и строками.

`контейнер::контейнер (InputIterator beg, InputIterator end)`

- Создает контейнер, инициализируемый всеми элементами из интервала [*beg, end*].
- Конструктор оформлен в виде шаблонной функции класса (см. с. 28). Это означает, что элементы исходного интервала могут относиться к произвольному типу, который может быть преобразован к типу элементов контейнера.

- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`контейнер::контейнер (InputIterator beg, InputIterator end,
const CompFunc& op)`

- Создает контейнер с критерием сортировки *op*, инициализируемый всеми элементами интервала *[beg,end]*.
- Конструктор оформлен в виде шаблонной функции класса (см. с. 28). Это означает, что элементы исходного интервала могут относиться к произвольному типу, который может быть преобразован к типу элементов контейнера.
- Критерий сортировки должен определять «строгую квазиупорядоченность» (см. с. 184).
- Поддерживается множествами, мульти множествами, отображениями, мультиотображениями.

`контейнер::~контейнер ()`

- Деструктор.
- Удаляет все элементы и освобождает память.
- Вызывает деструктор для каждого элемента.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

Немодифицирующие операции

Операции, связанные с размером

`size_type контейнер::size () const`

- Возвращает текущее количество элементов.
- Для проверки отсутствия элементов в контейнере используйте функцию `empty()`, потому что она может работать быстрее.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`bool контейнер::empty () const`

- Проверяет отсутствие элементов в контейнере.
- Эквивалент следующей команде, но может работать быстрее (особенно со списками):

`контейнер::size()==0`

- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`size_type контейнер::max_size () const`

- Возвращает максимальное количество элементов, которые могут храниться в контейнере.
- Возвращаемое значение зависит от модели памяти контейнера. Поскольку векторы обычно используют только один блок памяти, у них это значение может быть меньше, чем у других контейнеров.
- Поддерживается векторами, деками, списками, множествами, мультимножествами, отображениями, мультиотображениями, строками.

Операции, связанные с емкостью

`size_type контейнер::capacity () const`

- Возвращает количество элементов, которые могут храниться в контейнере без перераспределения памяти.
- Поддерживается векторами и строками.

`void контейнер::reserve(size_type num)`

- Резервирует внутреннюю память минимум для *num* элементов.
- Если *num* меньше текущей емкости контейнера, для векторов вызов игнорируется, а для строк является необязательным запросом на сокращение объема памяти.
- Методика уменьшения емкости векторов продемонстрирована в примере на с. 158.
- Каждое перераспределение памяти требует времени, а все ссылки, указатели и итераторы становятся недействительными. Функция `reserve()` может ускорить работу программы и сохранить действительными ссылки, указатели и итераторы (см. с. 158).
- Поддерживается векторами и строками.

Операции сравнения

`bool сравнение (const контейнер& c1, const контейнер& c2)`

- Возвращает результат сравнения двух контейнеров одного типа.
- Здесь *сравнение* — одна из следующих операций:

`operator ==
operator !=
operator <
operator >
operator <=
operator >=`

- Два контейнера считаются равными, если они содержат одинаковое количество элементов, если элементы попарно совпадают и следуют в одинаковом порядке (то есть результат проверки на равенство двух элементов в одинаковых позициях всегда равен `true`).

- Отношение «меньше/больше» между контейнерами проверяется по лексикографическому критерию (см. с. 356). Лексикографический критерий рассматривается при описании алгоритма `lexicographical_compare()` на с. 356.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

Специальные операции над ассоциативными контейнерами

Функции, перечисленные ниже, представляют собой специализированные реализации соответствующих алгоритмов STL, описанных на с. 336 и 388. Они превосходят унифицированные алгоритмы по скорости, поскольку в них учитывается тот факт, что элементы ассоциативных контейнеров автоматически сортируются. На практике эти функции обеспечивают логарифмическую сложность вместо линейной. Например, при поиске в контейнере, содержащем 1000 элементов, в среднем требуется не более 10 сравнений (см. с. 37).

`size_type контейнер::count (const T& value) const`

- Возвращает количество элементов со значением, равным `value`.
- Специализированная версия алгоритма `count()`, описанного на с. 337.
- `T` — тип сортируемых данных:
 - для множеств и мульти множеств — тип элемента;
 - для отображений и мультиотображений — тип ключа.
- Сложность линейная.
- Поддерживается множествами, мульти множествами, отображениями, мультиотображениями.

`iterator контейнер::find (const T& value)`

`const_iterator контейнер::find (const T& value) const`

- Обе версии возвращают позицию первого элемента со значением, равным `value`.
- Если элемент не найден, возвращается `end()`.
- Специализированные версии алгоритма `find()`, описанного на с. 340.
- `T` — тип сортируемых данных:
 - для множеств и мульти множеств — тип элемента;
 - для отображений и мультиотображений — тип ключа.
- Сложность логарифмическая.
- Поддерживаются множествами, мульти множествами, отображениями, мультиотображениями.

`iterator контейнер::lower_bound (const T& value)`

`const_iterator контейнер::lower_bound (const T& value) const`

- Обе версии возвращают первую позицию, в которой была бы вставлена копия `value` в соответствии с критерием сортировки.
- Возвращаемое значение представляет позицию первого элемента со значением, равным или большим `value` (или `end()`).

- Если элемент не найден, обе версии возвращают `end()`.
- Специализированные версии алгоритма `lower_bound()`, описанного на с. 402.
- Т — тип сортируемых данных:
 - для множеств и мульти множеств — тип элемента;
 - для отображений и мультиотображений — тип ключа.
- Сложность логарифмическая.
- Поддерживаются множествами, мульти множествами, отображениями, мультиотображениями.

```
iterator контейнер::upper_bound (const T& value)
const_iterator контейнер::upper_bound (const T& value) const
```

- Обе версии возвращают последнюю позицию, в которой была бы вставлена копия `value` в соответствии с критерием сортировки.
- Возвращаемое значение представляет позицию первого элемента со значением, большим `value` (или `end()`).
- Если элемент не найден, обе версии возвращают `end()`.
- Специализированные версии алгоритма `upper_bound()`, описанного на с. 402.
- Т — тип сортируемых данных:
 - для множеств и мульти множеств — тип элемента;
 - для отображений и мультиотображений — тип ключа.
- Сложность логарифмическая.
- Поддерживаются множествами, мульти множествами, отображениями, мультиотображениями.

```
pair<iterator, iterator> контейнер::equal_range (const T& value)
pair<const_iterator, const_iterator>
    контейнер::equal_range (const T& value) const
```

- Обе версии возвращают первую и последнюю позиции, в которых была бы вставлена копия `value` в соответствии с критерием сортировки.
- Возвращаемое значение определяет интервал элементов со значением, равным `value`.
- Результат вызова эквивалентен следующему:


```
make_pair(lower_bound(value), upper_bound(value))
```
- Специализированные версии алгоритма `equal_range()`, описанного на с. 404.
- Т — тип сортируемых данных:
 - для множеств и мульти множеств — тип элемента;
 - для отображений и мультиотображений — тип ключа.
- Сложность логарифмическая.
- Поддерживаются множествами, мульти множествами, отображениями, мультиотображениями.

`key_compare` контейнер::`key_comp` ()

- Возвращает критерий сравнения.
- Поддерживается множествами, мульти множествами, отображениями, мультиотображениями.

`value_compare` контейнер::`value_comp` ()

- Возвращает объект, используемый в качестве критерия сравнения.
- Для множеств и мульти множеств — эквивалент `key_comp`.
- Для отображений и мультиотображений — вспомогательный класс для критерия сравнения, при котором сравниваются только ключевые части двух элементов.
- Поддерживается множествами, мульти множествами, отображениями, мультиотображениями.

Присваивание

`контейнер& контейнер::operator= (const контейнер& c)`

- Присваивает контейнеру все элементы `c`; иначе говоря, все существующие элементы замещаются копиями элементов `c`.
- Оператор вызывает оператор присваивания для перезаписываемых элементов, копирующий конструктор для присоединяемых элементов или деструктор типа элемента для удаляемых элементов.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`void контейнер::assign (size_type num, const T& value)`

- Присваивает контейнеру `num` экземпляров `value`; иначе говоря, все существующие элементы замещаются `num` копиями `value`.
- `T` — тип элементов контейнера.
- Поддерживается векторами, деками, списками, строками.

`void контейнер::assign (InputIterator beg, InputIterator end)`

- Присваивает контейнеру содержимое интервала `[beg,end)`; иначе говоря, все существующие элементы замещаются копиями элементов `[beg,end)`.
- Функция оформлена в виде шаблонной функции класса (см. с. 28). Это означает, что элементы исходного интервала могут относиться к произвольному типу, который может быть преобразован к типу элементов контейнера.
- Поддерживается векторами, деками, списками, строками.

`void контейнер::swap (контейнер& c)`

- Меняет местами содержимое контейнера с содержимым контейнера `c`.
- В обоих контейнерах меняются как элементы, так и критерии сортировки.

- Функция выполняется с постоянной сложностью. Всегда используйте ее вместо присваивания, если присвоенный объект вам больше не нужен (см. с. 156).
- Для ассоциативных контейнеров функция генерирует исключения только в том случае, если эти исключения происходят при копировании или присваивании критерия сравнения. Для остальных контейнеров функция не генерирует исключения.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

```
void swap (контейнер& c1, контейнер& c2)
```

- Эквивалент следующего вызова (см. предыдущее описание):

```
c1.swap(c2)
```

- Для ассоциативных контейнеров функция генерирует исключения только в том случае, если эти исключения происходят при копировании или присваивании критерия сравнения. Для остальных контейнеров функция не генерирует исключений.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

Прямой доступ к элементам

```
reference контейнер::at (size_type idx)
const_reference контейнер::at (size_type idx) const
```

- Обе версии возвращают элемент с индексом *idx* (первому элементу соответствует индекс 0).
- Передача недействительного индекса (отрицательного, большего или равного `size()`) генерирует исключение `out_of_range`.
- Полученная ссылка может стать недействительной из-за последующих модификаций или перераспределения памяти.
- Если вы уверены в правильности индекса, используйте оператор [] — он работает быстрее.
- Поддерживаются векторами, деками и строками.

```
reference контейнер::operator [] (size_type idx)
const_reference контейнер::operator [] (size_type idx) const
```

- Обе версии возвращают элемент с индексом *idx* (первому элементу соответствует индекс 0).
- Передача недействительного индекса (отрицательного, большего или равного `size()`) приводит к непредсказуемым последствиям. При вызове необходимо быть твердо уверенными в правильности индекса; в противном случае следует использовать функцию `at()`.

- Ссылки, возвращаемые для неконстантных строк, могут стать недействительными из-за последующих модификаций строк или перераспределения памяти (см. с. 470).

- Поддерживаются векторами, деками и строками.

`T& map::operator [] (const key_type& key)`

- Оператор [] для ассоциативных массивов.

- Возвращает значение, соответствующее ключу *key* в отображении.

- Если в множестве нет ни одного элемента с ключом *key*, операция автоматически *создает* новый элемент со значением, инициализируемым конструктором по умолчанию соответствующего типа. Это означает, что индекс в принципе не может быть недействительным. Пример:

```
map<int, string> coll;
coll[77] = "hello"; // Вставка ключа 77 со значением "hello"
cout << coll[42]; // Ошибка - вставка ключа 42 со значением ""
// и вывод значения
```

За подробностями обращайтесь к с. 212.

- *T* – тип значения элемента.

- Эквивалент следующего вызова:

```
(*((insert(make_pair(x,T()))).first)).second
```

- Поддерживается отображениями.

```
reference контейнер::front ()
const_reference контейнер::front () const
```

- Обе версии возвращают первый элемент (элемент с индексом 0).

- Вызывающая сторона должна проследить за тем, чтобы контейнер содержал хотя бы один элемент (`size()>0`), иначе последствия вызова будут непредсказуемыми.

- Поддерживаются векторами, деками и списками.

```
reference контейнер::back ()
const_reference контейнер::back () const
```

- Обе версии возвращают последний элемент (элемент с индексом `size()-1`).

- Вызывающая сторона должна проследить за тем, чтобы контейнер содержал хотя бы один элемент (`size()>0`), иначе последствия вызова будут непредсказуемыми.

- Поддерживаются векторами, деками и списками.

Операции получения итераторов

Здесь описаны функции, возвращающие итераторы для перебора элементов контейнера. В табл. 6.34 перечислены категории итераторов (см. с. 257) для разных типов контейнеров.

Таблица 6.34. Категории итераторов для разных типов контейнеров

Контейнер	Категория итераторов
Вектор	Итератор произвольного доступа
Дек	Итератор произвольного доступа
Список	Двунаправленный итератор
Множество	Двунаправленный итератор, константные элементы
Мультимножество	Двунаправленный итератор, константные элементы
Отображение	Двунаправленный итератор, константные ключи
Мультиотображение	Двунаправленный итератор, константные ключи
Строка	Итератор произвольного доступа

```
iterator контейнер::begin ()
const_iterator контейнер::begin () const
```

- Обе версии возвращают итератор, установленный в начало контейнера (в позицию первого элемента).
- Если контейнер пуст, вызов эквивалентен следующему:

```
контейнер::end()
```

- Поддерживаются векторами, деками, списками, множествами, мультимножествами, отображениями, мультиотображениями, строками.

```
iterator контейнер::end ()
const_iterator контейнер::end () const
```

- Обе версии возвращают итератор, установленный в конец контейнера (в позицию за последним элементом).
- Если контейнер пуст, вызов эквивалентен следующему:

```
контейнер::begin()
```

- Поддерживаются векторами, деками, списками, множествами, мультимножествами, отображениями, мультиотображениями, строками.

```
reverse_iterator контейнер::rbegin ()
const_reverse_iterator контейнер::rbegin () const
```

- Обе версии возвращают обратный итератор, установленный в начало последовательности обратного перебора элементов (то есть в позицию последнего элемента контейнера).

- Если контейнер пуст, вызов эквивалентен следующему:

```
контейнер::rend()
```

- За дополнительной информацией об обратных итераторах обращайтесь на с. 270.

- Поддерживаются векторами, деками, списками, множествами, мультимножествами, отображениями, мультиотображениями, строками.

```
reverse_iterator контейнер::rend()
const_reverse_iterator контейнер::rend() const
```

- Обе версии возвращают обратный итератор, установленный в конец последовательности обратного перебора элементов (то есть в позицию перед первым элементом контейнера).
- Если контейнер пуст, вызов эквивалентен следующему:
`контейнер::rbegin()`
- За дополнительной информацией об обратных итераторах обращайтесь на с. 270.
- Поддерживаются векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

Вставка и удаление элементов

```
iterator контейнер::insert (const T& value)
pair<iterator,bool> контейнер::insert (const T& value)
```

- Обе версии вставляют копию *value* в ассоциативный контейнер.
- Первая сигнатура используется контейнерами, допускающими присутствие дубликатов (множествами и мультиотображениями). В этом случае возвращается позиция нового элемента.
- Вторая сигнатура используется контейнерами, в которых дубликаты запрещены (множествами и отображениями). Если вставить значение не удастся из-за того, что элемент с равным значением или ключом уже существует, возвращается позиция существующего элемента и *false*. Если вставка прошла успешно, возвращается позиция нового элемента и *true*.
- *T* — тип элементов контейнера. Для отображений и мультиотображений представляет пару «ключ/значение».
- Функции либо выполняются успешно, либо не вносят изменений.
- Поддерживаются множествами, мульти множествами, отображениями, мультиотображениями.

```
iterator контейнер::insert (iterator pos, const T& value)
```

- Вставляет копию *value* в позицию итератора *pos*.
- Возвращает позицию нового элемента.
- Для ассоциативных контейнеров (множеств, мульти множеств, отображений, мультиотображений) позиция используется только как рекомендация для начала поиска. Если вставка производится сразу же после *pos*, функция выполняется с амортизированной постоянной сложностью; в противном случае она выполняется с логарифмической сложностью.
- Если контейнером является множество или отображение, которое уже содержит элемент (или ключ), равный *value*, то функция не вносит изменений и возвращает позицию существующего элемента.

- В векторах и деках в результате выполнения операции итераторы и ссылки на другие элементы могут стать недействительными.
- Т – тип элементов контейнера. Для отображений и мультиотображений представляет пару «ключ/значение».
- Для строк *value* не передается по ссылке.
- В векторах и деках, если операции копирования (копирующий конструктор и оператор присваивания) не генерируют исключений, функция либо выполняется успешно, либо не вносит изменений. В остальных стандартных контейнерах функция либо выполняется успешно, либо не вносит изменений.
- Поддерживается векторами, деками, списками, множествами, мультимножествами, отображениями, мультиотображениями, строками.

```
void контейнер::insert (iterator pos, size_type num, const T& value)
```

- Вставляет *num* копий *value* в позицию итератора *pos*.
- В векторах и деках в результате выполнения операции итераторы и ссылки на другие элементы могут стать недействительными.
- Т – тип элементов контейнера. Для отображений и мультиотображений представляет пару «ключ/значение».
- Для строк *value* не передается по ссылке.
- В векторах и деках, если операции копирования (копирующий конструктор и оператор присваивания) не генерируют исключений, функция либо выполняется успешно, либо не вносит изменений. В списках функция либо выполняется успешно, либо не вносит изменений.
- Поддерживается векторами, деками, списками, строками.

```
void контейнер::insert (InputIterator beg, InputIterator end)
```

- Вставляет копии всех элементов в интервале *[beg,end)* в ассоциативный контейнер.
- Функция оформлена в виде шаблонной функции класса (см. с. 28). Это означает, что элементы исходного интервала могут относиться к произвольному типу, который может быть преобразован к типу элементов контейнера.
- Поддерживается множествами, мультимножествами, отображениями, мультиотображениями.

```
void контейнер::insert (iterator pos, InputIterator beg, InputIterator end)
```

- Вставляет копии всех элементов в интервале *[beg,end)* в позицию итератора *pos*.
- Функция оформлена в виде шаблонной функции класса (см. с. 28). Это означает, что элементы исходного интервала могут относиться к произвольному типу, который может быть преобразован к типу элементов контейнера.
- В векторах и деках в результате выполнения операции итераторы и ссылки на другие элементы могут стать недействительными.
- В списках функция либо выполняется успешно, либо не вносит изменений.
- Поддерживается векторами, деками, списками, строками.

```
void контейнер::push_front (const T& value)
```

- Вставляет копию *value* как новый первый элемент.

- Т — тип элементов контейнера.

- Эквивалент следующего вызова:

```
insert(begin(), value)
```

- В деках в результате выполнения операции итераторы других элементов становятся недействительными. Ссылки на другие элементы остаются действительными.

- Функция либо выполняется успешно, либо не вносит изменений.

- Поддерживается деками и списками.

```
void контейнер::push_back (const T& value)
```

- Вставляет копию *value* как новый последний элемент.

- Т — тип элементов контейнера.

- Эквивалент следующего вызова:

```
insert(end(), value).
```

- В векторах выполнение операции может привести к перераспределению памяти, в результате чего итераторы и ссылки на другие элементы становятся недействительными.

- В деках в результате выполнения операции итераторы других элементов становятся недействительными. Ссылки на другие элементы остаются действительными.

- Функция либо выполняется успешно, либо не вносит изменений.

- Поддерживается векторами, деками, списками, строками.

```
void::список::remove (const T& value)
```

```
void список::remove_if (UnaryPredicate op)
```

- Функция *remove()* удаляет все элементы со значением *value*.

- Функция *remove_if()* удаляет все элементы, для которых унарный предикат *op(элемент)* возвращает *true*.

- Предикат *op* не должен изменять свое состояние во время вызова функции.
За подробностями обращайтесь на с. 303.

- Обе версии вызывают деструкторы для удаляемых элементов.

- Порядок остальных элементов сохраняется.

- Это специализированная версия алгоритма *remove()* (см. с. 371), оптимизированная для списков.

- Т — тип элементов контейнера.

- Подробности и примеры приведены на с. 179.

- Функции генерируют исключения только при сравнении элементов.

- Поддерживаются списками.

`size_type контейнер::erase (const T& value)`

- Удаляет из ассоциативного контейнера все элементы со значением, равным `value`.
- Возвращает количество удаленных элементов.
- Вызывает деструкторы для удаляемых элементов.
- `T` – тип сортируемых данных:
 - для множеств и мультимножеств – тип элемента;
 - для отображений и мультиотображений – тип ключа.
- Функция не генерирует исключений.
- Поддерживается множествами, мультимножествами, отображениями, мультиотображениями.

`void контейнер::erase (iterator pos)`
`iterator контейнер::erase (iterator pos)`

- Обе версии удаляют элемент в позиции итератора `pos`.
- Для последовательных контейнеров (векторы, деки, списки и строки) используется вторая сигнатура, которая возвращает позицию следующего элемента (или `end()`).
- Для ассоциативных контейнеров (множества, мультимножества, отображения, мультиотображения) используется первая сигнатура, которая не имеет возвращаемого значения.
- Обе версии вызывают деструкторы для удаляемых элементов.
- Перед вызовом необходимо убедиться в том, что итератор `pos` имеет допустимое значение. Пример:

`coll.erase(coll.end()); // ОШИБКА - непредсказуемые последствия`

- В векторах и деках в результате выполнения операции итераторы и ссылки на другие элементы могут стать недействительными.
- В векторах и деках функция генерирует исключения только в том случае, если исключение генерируется копирующим конструктором или оператором присваивания. В остальных контейнерах функция не генерирует исключений.
- Поддерживается векторами, деками, списками, множествами, мультимножествами, отображениями, мультиотображениями, строками.

`void контейнер::erase (iterator beg, iterator end)`
`iterator контейнер::erase (iterator beg, iterator end)`

- Обе версии удаляют элементы в интервале `[beg,end)`.
- Для последовательных контейнеров (векторы, деки, списки и строки) используется вторая сигнатура, которая возвращает позицию элемента, находящегося за последним удаленным элементом (или `end()`).
- Для ассоциативных контейнеров (множества, мультимножества, отображения, мультиотображения) используется первая сигнатура, которая не имеет возвращаемого значения.

- Интервал является открытым, то есть удаляются все элементы, включая *beg*, но исключая *end*.
- Обе версии вызывают деструкторы для удаляемых элементов.
- Перед вызовом необходимо убедиться в том, что итераторы *beg* и *end* определяют действительный интервал элементов контейнера.
- В векторах и деках в результате выполнения операции итераторы и ссылки на другие элементы могут стать недействительными.
- В векторах и деках функция генерирует исключения только в том случае, если исключение генерируется копирующим конструктором или оператором присваивания. В остальных контейнерах функция не генерирует исключений.
- Поддерживается векторами, деками, списками, множествами, мультимножествами, отображениями, мультиотображениями, строками.

```
void контейнер::pop_front (const T& value)
```

- Удаляет первый элемент контейнера.
- Эквивалентно следующему вызову:

```
контейнер.erase(контейнер.begin()).
```

- Если контейнер не содержит элементов, последствия вызова непредсказуемы. Следовательно, при вызове необходимо убедиться в том, что контейнер содержит по крайней мере один элемент (`size()>0`).
- Функция не генерирует исключений.

- Поддерживается деками и списками.

```
void контейнер::pop_back (const T& value)
```

- Удаляет последний элемент.
- Эквивалентно следующему вызову, но при условии, что это выражение действительно (для векторов выполнение этого условия не гарантировано — см. с. 264):

```
контейнер.erase(--контейнер.end())
```

- Если контейнер не содержит элементов, последствия вызова непредсказуемы. Следовательно, при вызове необходимо убедиться в том, что контейнер содержит по крайней мере один элемент (`size()>0`).
- Функция не генерирует исключений,
- Поддерживается векторами, деками, списками.

```
void контейнер::resize (size_type num)
```

```
void контейнер::resize (size_type num, T value)
```

- Обе версии изменяют количество элементов до *num*.
- Если при вызове `size()==num`, вызов игнорируется.
- Если при вызове `size()<num`, дополнительные элементы создаются и присоединяются к концу контейнера. В первой версии новые элементы создаются

вызовом конструктора по умолчанию, во второй версии они создаются как копии *value*.

- Если при вызове `size() > n`, элементы в конце контейнера удаляются до заданного размера. В этом случае для удаляемых элементов вызываются деструкторы.
- В векторах и деках в результате выполнения операции итераторы и ссылки на другие элементы могут стать недействительными.
- В векторах и деках функция либо выполняется успешно, либо не вносит изменений, если при выполнении копирующего конструктора или оператора присваивания не было сгенерировано исключение. В списках функция либо выполняется успешно, либо не вносит изменений.
- Поддерживается векторами, деками, списками, строками.

```
void контейнер::clear () .
```

- Удаляет все элементы, оставляя контейнер пустым.
- Вызывает деструкторы для удаляемых элементов.
- Все итераторы и ссылки на контейнер становятся недействительными.
- В векторах и деках функция генерирует исключения только в том случае, если исключение генерируется копирующим конструктором или оператором присваивания. В остальных контейнерах функция не генерирует исключений.
- Поддерживается векторами, деками, списками, множествами, мультимножествами, отображениями, мультиотображениями, строками.

Специальные функции для списков

```
void список::unique ()
```

```
void список::unique (BinaryPredicate op)
```

- Обе версии оставляют в списке лишь один элемент с каждым значением и удаляют все последующие дубликаты. В результате значение каждого элемента списка отличается от значения всех остальных элементов.
- Первая версия удаляет все элементы, значение которых совпадает со значением одного из предшествующих элементов.
- Вторая версия удаляет все элементы, следующие после элемента *e*, для которых бинарный предикат *op(элемент, e)* возвращает `true`¹. Другими словами, предикат используется не для сравнения элемента с предшественником; элемент сравнивается с предыдущим элементом, который не был удален.
- Предикат *op* не должен изменять свое состояние во время вызова функции. За подробностями обращайтесь на с. 303.

¹ Вторая версия `sort()` работоспособна только в системах с поддержкой шаблонных функций классов (см. с. 28).

- Обе версии вызывают деструкторы для удаляемых элементов.
- Это специализированные версии алгоритма `unique()` (см. с. 375), оптимизированные для списков.
- Функции генерируют исключения только в том случае, если исключения генерируются при сравнении элементов.
- Поддерживаются списками.

`void список::splice (iterator pos, список& source)`

- Перемещает все элементы `source` в `*this` и вставляет их в позиции итератора `pos`.
- После вызова список `source` остается пустым.
- Если `source` и `*this` определяют одно и то же место в памяти, вызов приводит к непредсказуемым последствиям. Следовательно, при вызове необходимо убедиться в том, что список `source` отличается от текущего списка. Для перемещения элементов внутри списка применяется следующая форма функции `splice()`.
- Перед вызовом необходимо убедиться в том, что итератор `pos` имеет допустимое значение; в противном случае вызов приводит к непредсказуемым последствиям.
- Функция не генерирует исключений.

`void список::splice (iterator pos, список& source, iterator sourcePos)`

- Перемещает элемент в позиции `sourcePos` списка `source` в `*this` и вставляет его в позиции итератора `pos`.
- Список `source` и указатель `*this` могут определять одно и то же место в памяти. В этом случае элемент перемещается внутри списка.
- Если `source` — другой список, после выполнения операции он содержит на один элемент меньше.
- Перед вызовом необходимо убедиться в том, что `pos` представляет действительную позицию в `*this`, `sourcePos` является действительным итератором `source`, а значение `sourcePos` отлично от `source.end()`; в противном случае вызов приводит к непредсказуемым последствиям.
- Функция не генерирует исключений.

`void список::splice (iterator pos, список& source,
iterator sourceBeg, iterator sourceEnd)`

- Перемещает элементы из интервала `[sourceBeg,sourceEnd)` внутри списка `source` в `*this` и вставляет их в позицию итератора `pos`.
- Список `source` и указатель `*this` могут определять одно и то же место в памяти. В этом случае позиция `pos` не должна быть частью перемещаемого интервала, а элементы перемещаются внутри списка.
- Если `source` — другой список, после выполнения операции количество элементов в нем уменьшается.

- Перед вызовом необходимо убедиться в том, что *pos* представляет действительную позицию в **this*, а *sourceBeg* и *sourceEnd* определяют действительный интервал в *source*; в противном случае вызов приводит к непредсказуемым последствиям.

- Функция не генерирует исключений.

```
void список::sort ()
void список::sort (CompFunc op)
```

- Обе версии сортируют элементы списка.

- Первая версия сортирует все элементы оператором <.

- Вторая форма сортирует все элементы списка, сравнивая элементы функцией *op*¹:

```
op(elem1, elem2)
```

- Порядок следования элементов с равными значениями сохраняется (если при вызове не было сгенерировано исключение).

- Это специализированные версии алгоритмов *sort()* и *stable_sort()* (см. с. 389), оптимизированные для списков.

```
void список::merge (список& source)
void список::merge (список& source, CompFunc op)
```

- Обе версии выполняют слияние элементов списков *source* и **this*.

- После вызова список *source* остается пустым.

- Если перед вызовом списки **this* и *source* отсортированы по критерию < или *op*, итоговый список тоже будет отсортирован. Строго говоря, стандарт требует, чтобы оба списка были отсортированы перед вызовом, но на практике слияние также может выполняться и для несортированных списков. Тем не менее проведите дополнительную проверку, прежде чем выполнять слияние несортированных списков в своих программах.

- В первой форме используется оператор < в качестве критерия сортировки.

- Во второй форме используется необязательный критерий сортировки *op* для сравнения двух элементов²:

```
op(elem1, elem2)
```

- Это специализированные версии алгоритма *merge()* (см. с. 406), оптимизированные для списков.

```
void список::reverse ()
```

- Переставляет элементы списка в обратном порядке.

¹ Вторая версия *unique()* работоспособна только в системах с поддержкой шаблонных функций классов (см. с. 28).

² Вторая версия *merge()* работает способна только в системах с поддержкой шаблонных функций классов (см. с. 28).

- Это специализированная версия алгоритма `reverse()` (см. с. 379), оптимизированная для списков.
- Функция не генерирует исключений.

Поддержка распределителей памяти

Все контейнеры STL могут использоваться со специальной моделью памяти, определяемой объектом-распределителем (см. главу 15). Здесь перечислены функции, обеспечивающие поддержку распределителей.

В спецификации стандартных контейнеров обязательно требуется, чтобы все экземпляры типа распределителя были взаимозаменяемыми. Таким образом, память, выделенная одним экземпляром, может быть освобождена другим экземпляром того же типа. Следовательно, перемещение элементов (и перераспределение занимаемой ими памяти) между контейнерами одного типа не создает проблем.

Основные типы и функции распределителей

`контейнер::allocator_type`

- Тип распределителя.
- Поддерживается векторами, деками, списками, множествами, мультимножествами, отображениями, мультиотображениями, строками.

`allocator_type контейнер::get_allocator () const`

- Возвращает модель памяти контейнера.
- Поддерживается векторами, деками, списками, множествами, мультимножествами, отображениями, мультиотображениями, строками.

Конструкторы с передачей необязательных параметров

`explicit контейнер::контейнер (const Allocator& alloc)`

- Создает новый пустой контейнер, использующий модель памяти `alloc`.
- Поддерживается векторами, деками, списками, множествами, мультимножествами, отображениями, мультиотображениями, строками.

`контейнер::контейнер (const CompareFunc& op, const Allocator& alloc)`

- Создает новый пустой контейнер с критерием сортировки `op`, использующий модель памяти `alloc`.
- Критерий сортировки должен определять «строгую квазиупорядоченность» (см. с. 184).
- Поддерживается множествами, мультимножествами, отображениями, мультиотображениями.

`контейнер::контейнер (size_type num, const T& value, const Allocator& alloc)`

- Создает контейнер с `num` элементами, использующий модель памяти `alloc`.
- Элементы создаются как копии `value`.

- Т – тип элементов контейнера. Для строк *value* передается по значению.
- Поддерживается векторами, деками, списками, строками.

`контейнер::контейнер (InputIterator beg, InputIterator end,
· const Allocator& alloc)`

- Создает контейнер, инициализированный всеми элементами из интервала *[beg,end)* и использующий модель памяти *alloc*.
- Функция оформлена в виде шаблонной функции класса (см. с. 28). Это означает, что элементы исходного интервала могут относиться к произвольному типу, который может быть преобразован к типу элементов контейнера.
- Поддерживается векторами, деками, списками, множествами, мульти множествами, отображениями, мультиотображениями, строками.

`контейнер::контейнер (InputIterator beg, InputIterator end,
const CompFunc& op, const Allocator& alloc)`

- Создаст контейнер с критерием сортировки *op*, инициализированный всеми элементами из интервала *[beg,end)* и использующий модель памяти *alloc*.
- Функция оформлена в виде шаблонной функции класса (см. с. 28). Это означает, что элементы исходного интервала могут относиться к произвольному типу, который может быть преобразован к типу элементов контейнера.
- Критерий сортировки должен определять «строгую квазиупорядоченность» (см. с. 184).
- Поддерживается множествами, мульти множествами, отображениями, мультиотображениями.

Обработка исключений в контейнерах STL

Как упоминалось на с. 148, контейнеры предоставляют разный уровень гарантий в отношении исключений. Стандартная библиотека C++ гарантирует отсутствие утечки ресурсов и нарушения контейнерных инвариантов в отношении исключения, однако некоторые операции предоставляют более твердые гарантии (при условии, что аргументы удовлетворяют некоторым условиям): они могут гарантировать транзакционную безопасность (принятие/откат) и даже то, что они никогда не генерируют исключений. В табл. 6.35 перечислены все операции, для которых предоставляются такие гарантии¹.

Таблица 6.35. Контейнерные операции с особыми гарантиями в отношении исключений

Операция	Страница	Гарантия
<code>vector::push_back()</code>	247	Либо завершается успешно, либо не вносит изменений
<code>vector::insert()</code>	246	Либо завершается успешно, либо не вносит изменений, если при копировании/присваивании элементов не генерируются исключения

¹ Большое спасибо Грегу Колвину (Greg Colvin) и Дэйву Абрахамсу (Dave Abrahams) за предоставленную таблицу.

Операция	Страница	Гарантия
vector::pop_back()	249	Не генерирует исключений
vector::erase()	248	Не генерирует исключений, если они не генерируются при копировании/присваивании элементов
vector::clear()	250	Не генерирует исключений, если они не генерируются при копировании/присваивании элементов
vector::swap()	242	Не генерирует исключений
deque::push_back()	247	Либо завершается успешно, либо не вносит изменений
deque::push_front()	247	Либо завершается успешно, либо не вносит изменений
deque::insert()	246	Либо завершается успешно, либо не вносит изменений, если при копировании/присваивании элементов не генерируются исключения
deque::pop_back()	249	Не генерирует исключений
deque::pop_front()	249	Не генерирует исключений
deque::erase()		Не генерирует исключений, если они не генерируются при копировании/присваивании элементов
deque::clear()	250	Не генерирует исключений, если они не генерируются при копировании/присваивании элементов
deque::swap()	242	Не генерирует исключений
list::push_back()	247	Либо завершается успешно, либо не вносит изменений
list::push_front()	247	Либо завершается успешно, либо не вносит изменений
list::insert()	246	Либо завершается успешно, либо не вносит изменений
list::pop_back()	249	Не генерирует исключений
list::pop_front()	249	Не генерирует исключений
list::erase()	248	Не генерирует исключений
list::clear()	250	Не генерирует исключений
list::remove()	247	Не генерирует исключений, если они не генерируются при сравнении элементов
list::remove_if()	247	Не генерирует исключений, если они не генерируются предикатом
list::unique()	250	Не генерирует исключений, если они не генерируются при сравнении элементов
list::splice()	251	Не генерирует исключений
list::merge()	252	Либо завершается успешно, либо не вносит изменений, если при сравнении элементов не генерируются исключения
list::reverse()	252	Не генерирует исключений
list::swap()	242	Не генерирует исключений

Таблица 6.35 (продолжение)

Операция	Страница	Гарантия
[multi]set::insert()	246	При вставке одного элемента либо завершается успешно, либо не вносит изменений
[multi]set::erase()	248	Не генерирует исключений
[multi]set::clear()	250	Не генерирует исключений
[multi]set::swap()	242	Не генерирует исключений, если они не генерируются при копировании/присваивании критерия сравнения
[multi]map::insert()	246	При вставке одного элемента либо завершается успешно, либо не вносит изменений
[multi]map::erase()	248	Не генерирует исключений
[multi]map::clear()	250	Не генерирует исключений
[multi]map::swap()	242	Не генерирует исключений, если они не генерируются при копировании/присваивании критерия сравнения

Векторы, деки и списки также предоставляют особые гарантии для `resize()`. В соответствии с ними последствия от вызова `resize()` эквивалентны либо вызову `erase()`, либо вызову `insert()`, либо отсутствию операции:

```
void контейнер::resize (size_type num, T value = T())
{
    if (num > size()) {
        insert (end(), num-size(), value);
    }
    else if (num < size()) {
        erase (begin() + num, end());
    }
}
```

Иначе говоря, предоставляемые гарантии определяются комбинацией гарантий для `erase()` и `insert()` (см. с. 249).

7 Итераторы STL

Заголовочные файлы итераторов

Каждый контейнер сам определяет типы своих итераторов, поэтому для работы с итераторами контейнеров специальные заголовочные файлы не нужны. Тем не менее существует несколько специальных итераторов (например, обратных), которые определяются в заголовочном файле `<iterator>`¹, хотя этот файл напрямую включается в программы довольно редко. Он используется контейнерами для определения типов обратных итераторов и обычно подгружается ими автоматически.

Категории итераторов

Итератором называется объект, предназначенный для последовательного перебора элементов. Перебор осуществляется через единый интерфейс, основой для которого стал интерфейс обычных указателей (общие сведения об итераторах приводятся на с. 96). Итераторы подчиняются принципу чистой абстракции, то есть любой объект, который *ведет себя* как итератор, *является* итератором. Тем не менее итераторы обладают разными свойствами; иногда это очень существенно, поскольку для работы некоторых алгоритмов необходимы особые свойства итераторов. Например, для алгоритмов сортировки нужны итераторы произвольного доступа, поскольку без них эффективность алгоритмов была бы слишком низкой. Из-за этого итераторы делятся на несколько категорий (рис. 7.1). В табл. 7.1 приведены краткие характеристики этих категорий.

Таблица 7.1. Характеристики категорий итераторов

Категория	Возможности	Поддержка
Итератор ввода	Чтение в прямом направлении	Потоковый итератор ввода
Итератор вывода	Запись в прямом направлении	Потоковый итератор вывода, итератор вставки

продолжение ↓

¹ В исходной версии STL заголовочный файл итераторов назывался `<iterator.h>`.

Таблица 7.1 (продолжение)

Категория	Возможности	Поддержка
Прямой итератор	Чтение и запись в прямом направлении	
Двунаправленный итератор	Чтение и запись в прямом и обратном направлениях	Списки, множества, мультимножества, отображения, мультиотображения
Итератор произвольного доступа	Чтение и запись с произвольным доступом	Вектор, дек, строка, массив



Рис. 7.1. Категории итераторов

Итераторы ввода

Итератор ввода перемещается только вперед и поддерживает только чтение (то есть возвращает значения элементов в порядке их перебора). В табл. 7.2 перечислены операции итераторов ввода.

Таблица 7.2. Операции итераторов ввода

Выражение	Описание
*iter	Обращение к элементу
iter->member	Обращение к переменной или функции элемента
++iter	Смещение вперед (возвращает новую позицию)
iter++	Смещение вперед (возвращает старую позицию)
iter1 == iter2	Проверка двух итераторов на равенство
iter1 != iter2	Проверка двух итераторов на неравенство
TYPE(iter)	Копирование итератора (копирующий конструктор)

Итератор ввода читает элементы только один раз. Если скопировать итератор ввода и выполнить чтение через оригинал и копию, вы можете получить разные значения.

Практически любой итератор обладает базовыми возможностями итераторов ввода, но большинство итераторов способно на большее. Типичным примером «чистого» итератора ввода является итератор, читающий данные из стандартного входного потока данных (обычно с клавиатуры). Одно значение не может быть прочитано дважды. После чтения слова из входного потока данных при следующем обращении будет получено другое слово.

Два итератора ввода считаются равными, если они находятся в одной позиции. Однако, как упоминалось выше, это не означает, что при следующем чтении они вернут одно и то же значение.

По возможности используйте префиксный оператор перемещения итератора вместо постфиксного, потому что он может работать более эффективно. Дело в том, что префиксной форме не нужно возвращать старое значение, которое должно храниться во временном объекте. Следовательно, для итератора `pos` (и любого абстрактного типа данных) рекомендуется использовать префиксный оператор:

```
++pos // Правильно и быстро
```

Префиксный оператор работает быстрее постфиксного:

```
pos++ // Правильно, но не быстро
```

Это утверждение относится и к оператору `--` (для тех итераторов, для которых он определен, `--` итераторы ввода его не поддерживают).

Итераторы вывода

Итераторы вывода составляют пару с итераторами ввода. Они тоже перемещаются только вперед, но выполняют запись. Таким образом, присваивание новых значений выполняется только для отдельных элементов. Итератор вывода не может использоваться для повторного перебора интервала. Запись производится в некую абстрактную «черную дыру»; если вы повторно записываете данные в той же позиции в исходную «черную дыру», ничто не гарантирует, что они будут записаны поверх предыдущих данных. В табл. 7.3 перечислены операции итераторов вывода.

Таблица 7.3. Операции итераторов вывода

Выражение	Описание
<code>*iter=value</code>	Записывает <code>value</code> в позицию, определяемую итератором
<code>++iter</code>	Смещение вперед (возвращает новую позицию)
<code>iter++</code>	Смещение вперед (возвращает старую позицию)
<code>TYPE(iter)</code>	Копирование итератора (копирующий конструктор)

Обратите внимание: для итераторов вывода операции сравнения не нужны. Вам не удастся проверить, действителен ли итератор вывода и успешна ли состоялась «запись». Возможна только запись и ничего более.

На практике итераторы обычно поддерживают как чтение, так и запись. Абсолютное большинство итераторов обладает свойствами итераторов вывода (как

и итераторов ввода). Типичным примером «чистого» итератора вывода служит итератор для записи в стандартный выходной поток данных (например, на экран или на принтер). Если использовать два итератора вывода для записи на экран, то второе слово будет выводиться после первого, а не поверх него. Другой типичный пример итератора вывода — итератор вставки, предназначенный для занесения новых значений в контейнер. Операция присваивания приводит к тому, что в контейнер вставляется новый элемент. При последующей записи второе значение не стирает первое, а просто вставляется отдельно от него. Итераторы вставки рассматриваются на с. 275.

Прямые итераторы

Прямые итераторы представляют собой комбинацию итераторов ввода и вывода. Они обладают всеми свойствами итераторов ввода и большинством свойств итераторов вывода. Операции прямых итераторов перечислены в табл. 7.4.

Таблица 7.4. Операции прямых итераторов

Выражение	Описание
<code>*iter</code>	Обращение к элементу
<code>iter->member</code>	Обращение к переменной или функции элемента
<code>++iter</code>	Смещение вперед (возвращает новую позицию)
<code>iter++</code>	Смещение вперед (возвращает старую позицию)
<code>iter1 == iter2</code>	Проверка двух итераторов на равенство
<code>iter1 != iter2</code>	Проверка двух итераторов на неравенство
<code>TYPE()</code>	Копирование итератора (конструктор по умолчанию)
<code>TYPE(iter)</code>	Копирование итератора (копирующий конструктор)
<code>iter1 = iter2</code>	Присваивание итератора

В отличие от итераторов ввода и вывода прямые итераторы могут ссылаться на один и тот же элемент коллекции и обрабатывать его по несколько раз.

Возможно, вас интересует, почему прямой итератор не обладает всеми свойствами итератора вывода. Существует одно важное ограничение, из-за которого код, действительный для итераторов вывода, оказывается недействительным для прямых итераторов.

О *Итераторы вывода* позволяют записывать данные без проверки конца последовательности. Более того, вы даже не можете сравнить итератор вывода с конечным итератором, потому что итераторы вывода не поддерживают операцию сравнения. Следовательно, для итератора вывода `pos` следующий цикл правилен, а для прямого итератора — нет:

```
// OK для итераторов вывода
// ОШИБКА для прямых итераторов
while (true) {
```

```
*pos = foo();
++pos;
}
```

- При работе с *прямыми итераторами* перед разыменованием (обращением к данным) необходимо заранее убедиться в том, что это возможно. Приведенный выше цикл неправилен для прямых итераторов, поскольку он приводит к попытке обращения по итератору `end()` в конце коллекции с непредсказуемыми последствиями. Для прямых итераторов цикл необходимо изменить следующим образом:

```
// OK для прямых итераторов
// ОШИБКА для итераторов вывода
while (pos != coll.end()) {
    *pos = foo();
    ++pos;
}
```

Для итераторов вывода цикл не компилируется, потому что для них не определен оператор `!=`.

Двунаправленные итераторы

Двунаправленными итераторами называются прямые итераторы, поддерживающие возможность перебора элементов в обратном направлении. Для этой цели в них определяется дополнительный оператор `--` (табл. 7.5).

Таблица 7.5. Дополнительные операции для двунаправленных итераторов

Выражение	Описание
<code>--iter</code>	Смещение назад (возвращает новую позицию)
<code>iter--</code>	Смещение назад (возвращает старую позицию)

Итераторы произвольного доступа

Итераторами произвольного доступа называются двунаправленные итераторы, поддерживающие прямой доступ к элементам. Для этого в них определяются «вычисления с итераторами» (по аналогии с математическими вычислениями с обычными указателями) — вы можете складывать и вычитать смещения, обрабатывать разности и сравнивать итераторы при помощи операторов отношения (таких, как `<` и `>`). В табл. 7.6 перечислены дополнительные операции для итераторов произвольного доступа.

Итераторы произвольного доступа поддерживаются следующими объектами и типами:

- контейнеры с произвольным доступом (`vector`, `deque`);
- строки (`string`, `wstring`);
- обычные массивы (указатели).

Таблица 7.6. Дополнительные операции для итераторов произвольного доступа

Выражение	Описание
iter[n]	Обращение к элементу с индексом n
iter+=n	Смещение на n элементов вперед (или назад, если n<0)
iter-=n	Смещение на n элементов назад (или вперед, если n<0)
iter+n	Возвращает итератор для n-го элемента вперед от текущей позиции
n+iter	Возвращает итератор для n-го элемента вперед от текущей позиции
iter-n	Возвращает итератор для n-го элемента назад от текущей позиции
iter1-iter2	Возвращает расстояние между iter1 и iter2
iter1<iter2	Проверяет, что iter1 меньше iter2
iter1>iter2	Проверяет, что iter1 больше iter2
iter1<=iter2	Проверяет, что iter1 предшествует или совпадает с iter2
iter1>=iter2	Проверяет, что iter1 не предшествует iter2

Представленная ниже программа демонстрирует особые возможности итераторов произвольного доступа.

```
// iter/itercat.cpp
#include <vector>
#include <iostream>
using namespace std;

int main()
{
    vector<int> coll;

    // Вставка элементов от -3 до 9
    for (int i=-3; i<=9; ++i) {
        coll.push_back (i);
    }

    /* Вывод количества элементов как расстояния между
     * начальным и конечным итераторами
     * - ВНИМАНИЕ: применение оператора - для итераторов
     */
    cout << "number/distance: " << coll.end()-coll.begin() << endl;

    /* Вывод всех элементов
     * - ВНИМАНИЕ: применение оператора < вместо оператора !=
     */
    vector<int>::iterator pos;
    for (pos=coll.begin(); pos<coll.end(); ++pos) {
        cout << *pos << ' ';
    }
}
```

```
}

cout << endl;

/* Вывод всех элементов
 * - ВНИМАНИЕ: применение оператора [] вместо оператора *
 */
for (int i=0; i<coll.size(); ++i) {
    cout << coll.begin()[i] << ' ';
}
cout << endl;

/* Вывод каждого второго элемента
 * - ВНИМАНИЕ: применение оператора +=
 */
for (pos = coll.begin(); pos < coll.end()-1; pos += 2) {
    cout << *pos << ' ';
}
cout << endl;
}
```

Результат выполнения программы выглядит так:

```
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
```

Этот пример не работает со списками, множествами и отображениями, потому что все операции с пометкой **ВНИМАНИЕ:** поддерживаются только для итераторов произвольного доступа. В частности, помните, что оператор `<` может использоваться в критерии завершения цикла только для итераторов произвольного доступа.

Следующее выражение в последнем цикле требует, чтобы коллекция `coll` содержала минимум один элемент:

```
pos < coll.end()-1
```

Если коллекция пуста, то `coll.end()-1` будет ссылаться на позицию перед `coll.begin()`. Сравнение останется работоспособным, но, формально говоря, перемещение итератора в позицию перед началом коллекции приводит к непредсказуемым последствиям. То же самое происходит при выходе за пределы конечного итератора `end()` при выполнении выражения `pos+=2`. Таким образом, следующая формулировка последнего цикла очень опасна, потому что при четном количестве элементов в коллекции она приводит к непредсказуемым последствиям (рис. 7.2):

```
for (pos = coll.begin(); pos < coll.end(); pos += 2) {
    cout << *pos << ' ';
}
```

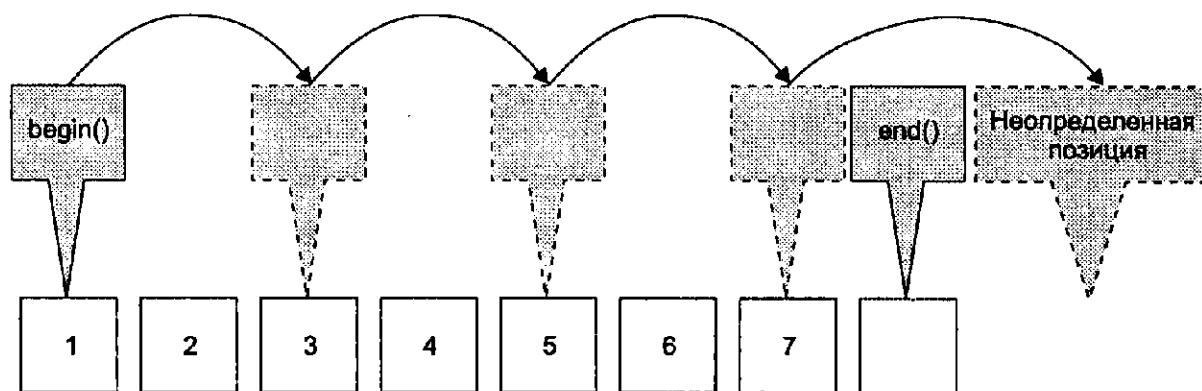


Рис. 7.2. Категории итераторов

Проблема увеличения и уменьшения итераторов в векторах

При применении в векторах операторов увеличения и уменьшения к итераторам возникает странная проблема. Вообще говоря, увеличение и уменьшение временных итераторов разрешено, но для векторов и строк оно обычно запрещено. Рассмотрим следующий пример:

```
std::vector<int> coll;
...
// Сортировка, начиная со второго элемента
// - НЕПЕРЕНОСИМАЯ версия
if (coll.size() > 1) {
    sort(++coll.begin(), coll.end());
}
```

Обычно компиляция строки с вызовом `sort()` завершается неудачей, но если заменить вектор деком, компиляция пройдет нормально. Иногда программа компилируется даже для векторов — это зависит от реализации класса `vector`.

Дело в том, что векторные итераторы обычно реализуются в виде обычных указателей. Для всех базовых типов данных, в том числе и для указателей, модификация временных значений запрещена. С другой стороны, для структур и классов она разрешена. Следовательно, если итератор реализован в виде обычного указателя, программа не компилируется; если итератор реализован в виде класса, компиляция проходит успешно. Для деков, списков, множеств и отображений такой проблемы не существует, поскольку в них итераторы в принципе не реализуются в виде обычных указателей, но для векторов все зависит от реализации. В большинстве реализаций задействованы обычные указатели. Но, например, в «безопасной» версии STL итераторы реализованы в виде классов. Чтобы приведенный выше фрагмент был переносимым, достаточно определить промежуточный объект:

```
std::vector<int> coll;
...
// Сортировка, начиная со второго элемента
// - ПЕРЕНОСИМАЯ версия
if (coll.size() > 1) {
```

```
    std::vector<int>::iterator beg = coll.begin();
    sort(++coll.beg, coll.end());
}
```

Проблема не так серьезна, как кажется на первый взгляд. Она обнаруживается на стадии компиляции и поэтому не приводит к непредсказуемым последствиям. С другой стороны, проблема достаточно нетривиальна, и на ее выявление иногда уходит немало времени. Кроме того, она относится не только к векторам, но и к строкам. Строковые итераторы тоже обычно реализуются в виде обычных указателей на символы, хотя это и не обязательно.

Вспомогательные функции итераторов

Стандартная библиотека C++ содержит три вспомогательные функции для работы с итераторами: `advance()`, `distance()` и `iter_swap()`. Первые две функции предоставляют для любого итератора возможности, которыми обычно обладают только итераторы произвольного доступа: перемещение итератора сразу на несколько элементов вперед (или назад) и вычисление разности между итераторами. Третья вспомогательная функция меняет местами элементы, на которые ссылаются два итератора.

Перебор итераторов функцией `advance`

Функция `advance()` перемещает итератор, передаваемый ей в качестве аргумента. При этом смещение может производиться в прямом (или обратном) направлении сразу на несколько элементов:

```
#include <iterator>
void advance (InputIterator& pos, Dist n)
```

- Перемещает итератор ввода `pos` на `n` элементов вперед (или назад).
- Для двунаправленных итераторов и итераторов произвольного доступа значение `n` может быть отрицательным (перемещение в обратном направлении).
- `Dist` — тип шаблона. Обычно является целым типом, поскольку для него вызываются такие операции, как `<`, `++`, `--` и сравнение с 0.
- Обратите внимание, что функция `advance()` не проверяет выход за пределы `end()` (и не может проверять, поскольку в общем случае итератор не располагает информацией о контейнере, с которым он работает). Таким образом, вызов этой функции может привести к непредсказуемым последствиям из-за вызова оператора `++` в конце последовательности.

Благодаря разным трактовкам итераторов (см. с. 288) функция всегда использует оптимальную реализацию в зависимости от категории итератора. Для итераторов произвольного доступа она просто вызывает `pos+=n`. Следовательно, для таких итераторов функция `advance()` имеет постоянную сложность. Для остальных итераторов используется n -кратный вызов `pos` (или `--pos`, если значение `n` отрицательно). Таким образом, для всех остальных категорий итераторов функция `advance()` выполняется с линейной сложностью.

Чтобы иметь возможность свободно менять типы контейнера и итератора, используйте функцию `advance()` вместо оператора `+=`. Однако следует помнить, что при этом возможно непредвиденное снижение быстродействия при переходе на другие типы контейнеров, не поддерживающие итераторы произвольного доступа (именно из-за снижения быстродействия оператор `+=` применяется только для итераторов произвольного доступа). Также помните, что функция `advance()` не возвращает значения, а оператор `+=` возвращает новую позицию и может входить в более сложное выражение. Пример использования функции `advance()`:

```
// iter/advance1.cpp
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // Вставка элементов от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    list<int>::iterator pos = coll.begin();

    // Вывод текущего элемента
    cout << *pos << endl;

    // Перемещение вперед на три элемента
    advance (pos, 3);

    // Вывод текущего элемента
    cout << *pos << endl;

    // Перемещение назад на один элемент
    advance (pos, -1);

    // Вывод текущего элемента
    cout << *pos << endl;
}
```

Вызовы `advance()` переводят итератор на три элемента вперед и на один элемент назад. Результат:

```
1
4
3
```

Существуют и другие применения функции `advance()`. Например, вы можете проигнорировать часть ввода от итераторов, читающих данные из входного потока. Пример приведен на с. 286.

Обработка расстояния между итераторами функцией *distance*

Функция *distance()* вычисляет разность 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 одного контейнера.
- Если итераторы не являются итераторами произвольного доступа, итератор *pos2* должен быть доступен от *pos1* (то есть должен находиться в той же или в одной из следующих позиций).
- Тип возвращаемого значения *Dist* определяет тип разности в соответствии с типом итераторов:

```
iterator_traits<InputIterator>::difference_type
```

Подробности приведены на с. 288.

При помощи итераторных тегов¹ функция *distance()* выбирает оптимальную реализацию в соответствии с категорией итератора. Для итераторов произвольного доступа она просто возвращает *pos2-pos1*; следовательно, для таких итераторов функция *distance()* имеет постоянную сложность. Для остальных категорий итераторов функция *distance()* последовательно увеличивает *pos1* до достижения *pos2*, после чего возвращается количество увеличений. В этом случае функция работает с линейной сложностью, и для таких категорий итераторов лучше воздержаться от ее использования. Таким образом, функция *distance()* обладает низким быстродействием для всех итераторов, кроме итераторов произвольного доступа.

Реализация функции *distance()* представлена на с. 289. Ниже приведен пример ее использования.

```
// iter/distance.cpp
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // Вставка элементов от -3 до 9
    for (int i=-3; i<=9; ++i) {
        coll.push_back(i);
    }
}
```

¹ См. с. 287 – Примеч. перев.

```

// Поиск элемента со значением 5
list<int>::iterator pos;
pos = find (coll.begin(), coll.end(),      // Интервал
            5);                           // Значение

if (pos != coll.end()) {
    // Вычисление и вывод расстояния от начала коллекции
    cout << "difference between beginning and 5: "
        << distance(coll.begin(), pos) << endl;
}
else {
    cout << "5 not found" << endl;
}
}

```

Функция `find()` присваивает итератору `pos` позицию элемента со значением 5. Функция `distance()` использует эту позицию для вычисления разности между позицией и началом коллекции. Результат выполнения программы выглядит так:

`difference between beginning and 5: 8`

Чтобы иметь возможность свободно менять тип контейнера и итератора, используйте функцию `distance()` вместо оператора `-`. Однако следует помнить, что переход от итераторов произвольного доступа на другие итераторы ухудшает быстродействие.

При вычислении разности между двумя итераторами, не являющимися итераторами произвольного доступа, необходима осторожность. Первый итератор должен ссылаться на элемент, предшествующий второму или совпадающий с ним. В противном случае попытка вычисления разности неизбежно приводит к неожиданным последствиям. Другие аспекты этой проблемы затрагиваются в комментариях по поводу поиска в подмножестве (см. с. 111).

В первых версиях STL использовалась другая сигнатура функции `distance()`. Вместо того чтобы возвращать разность, функция передавала ее в третьем аргументе. Эта версия была крайне неудобной, поскольку не позволяла напрямую вызывать функцию в выражениях. Если у вас устаревшая версия, воспользуйтесь простым обходным решением:

```

// iter/distance.hpp
template <class Iterator>
inline long distance (Iterator pos1, Iterator pos2)
{
    long d = 0;
    distance (pos1, pos2, d);
    return d;
}

```

В этой функции тип возвращаемого значения не зависит от реализации, а жестко кодируется в виде типа `long`. В общем случае типа `long` должно хватать для всех возможных значений, но это не гарантируется.

Перестановка элементов функцией `iter_swap`

Простая вспомогательная функция `iter_swap()` меняет местами элементы, на которые ссылаются два итератора:

```
#include <algorithm>
void iter_swap (ForwardIterator pos1, ForwardIterator pos2)
```

- Меняет местами элементы, на которые ссылаются итераторы `pos1` и `pos2`.
- Итераторы не обязаны относиться к одному типу. Тем не менее значения должны быть совместимы по присваиванию.

Простой пример использования функции `iter_swap()` (функция `PRINT_ELEMENTS()` представлена на с. 128):

```
// iter_swap1.cpp
#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
    list<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    PRINT_ELEMENTS(coll);

    // Перестановка первого и второго значений
    iter_swap (coll.begin(), ++coll.begin());

    PRINT_ELEMENTS(coll);

    // Перестановка первого и последнего значений
    iter_swap (coll.begin(), --coll.end());

    PRINT_ELEMENTS(coll);
}
```

Результат выполнения программы выглядит так:

```
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
```

Учтите, что с векторами эта программа обычно не работает. Дело в том, что операции `++coll.begin()` и `--coll.end()` выполняются с временными указателями (дополнительная информация на эту тему приведена на с. 264).

Итераторные адаптеры

В этом разделе рассматриваются итераторные адаптеры — специальные итераторы, позволяющие выполнять алгоритмы, которые поддерживают перебор элементов в обратном порядке, режим вставки и потоки данных.

Обратные итераторы

Обратные итераторы — адаптеры, переопределяющие операторы `++` и `--` так, что перебор элементов производится в обратном направлении. Следовательно, при использовании этих итераторов вместо обычных алгоритмы обрабатывают элементы в обратном порядке. Все стандартные контейнерные классы позволяют использовать обратные итераторы для перебора элементов. Рассмотрим следующий пример:

```
// iter/reviter1.cpp
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

void print (int elem)
{
    cout << elem << ' ';
}

int main()
{
    list<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // Вывод всех элементов в нормальном порядке
    for_each (coll.begin(), coll.end(),           // Интервал
              print);                           // Операция
    cout << endl;

    // Вывод всех элементов в обратном порядке
    for_each (coll.rbegin(), coll.rend(),         // Интервал
              print);                           // Операция
    cout << endl;
}
```

Функции `rbegin()` и `rend()` возвращают обратный итератор. По аналогии с функциями `begin()` и `end()` эти итераторы определяют набор обрабатываемых элемен-

тов в виде полуоткрытого интервала, однако они производят перебор в обратном направлении:

- `rbegin()` возвращает позицию первого элемента при обратном переборе (то есть позицию последнего элемента контейнера);
- `rend()` возвращает позицию за последним элементом при обратном переборе (то есть позицию *перед* первым элементом контейнера).

Итераторы и обратные итераторы

Обычный итератор можно преобразовать в обратный. Естественно, такой итератор должен быть двунаправленным, но следует помнить, что в процессе преобразования происходит смещение логической позиции итератора. Рассмотрим следующую программу:

```
// iter2/reviter2.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // Поиск позиции элемента со значением 5
    vector<int>::iterator pos;
    pos = find (coll.begin(), coll.end(),
                5);

    // Вывод значения, на которое ссылается итератор pos
    cout << "pos: " << *pos << endl;

    // Преобразование итератора в обратный итератор rpos
    vector<int>::reverse_iterator rpos(pos);

    // Вывод значения, на которое ссылается обратный итератор rpos
    cout << "rpos: " << *rpos << endl;
}
```

Программа выводит следующий результат:

```
pos: 5
rpos: 4
```

Итак, если вывести значение, на которое ссылается итератор, а затем преобразовать итератор в обратный, значение изменится. И это не ошибка — так

и должно быть! Такое поведение следует из того факта, что интервалы являются полуоткрытыми. Чтобы задать все элементы контейнера, необходимо установить второй итератор в позицию за последним элементом. Для обратного итератора эта позиция соответствует позиции перед первым элементом. Но, к сожалению, такая позиция может оказаться несуществующей. Контейнер не обязан гарантировать, что позиция перед его первым элементом является действительной. Кроме того, обычные строки и массивы тоже могут быть контейнерами, а язык не гарантирует, что адресация массива не начинается с нуля.

В итоге проектировщики обратных итераторов пошли на уловку: они «физически» обратили концепцию полуоткрытого интервала. Интервал, определяемый обратными итераторами, включает начало, но не включает конец. Тем не менее на логическом уровне обратные итераторы ведут себя, как обычные. Следовательно, существует различие между физической позицией, определяющей элемент, на который ссылается итератор, и логической позицией, определяющей значение, на которое ссылается итератор (рис. 7.3). Спрашивается, что должно происходить при преобразовании обычного итератора в обратный? Должен ли итератор сохранить свою логическую позицию (значение) или физическую позицию (элемент)? Как видно из предыдущего примера, выполняется второе условие, то есть логическая позиция смещается к предыдущему элементу (рис. 7.4).

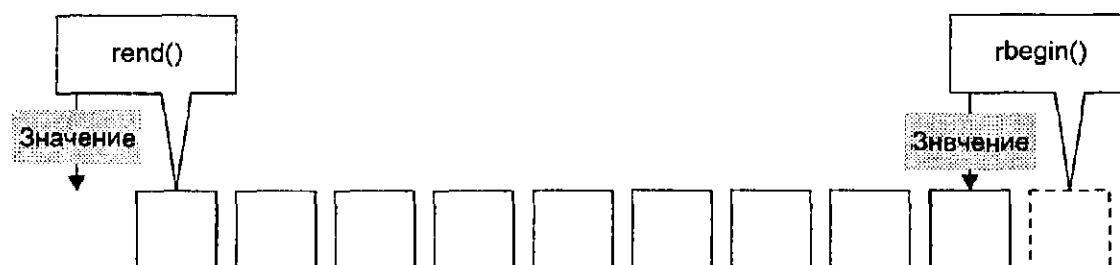


Рис. 7.3. Позиция и значение обратного итератора

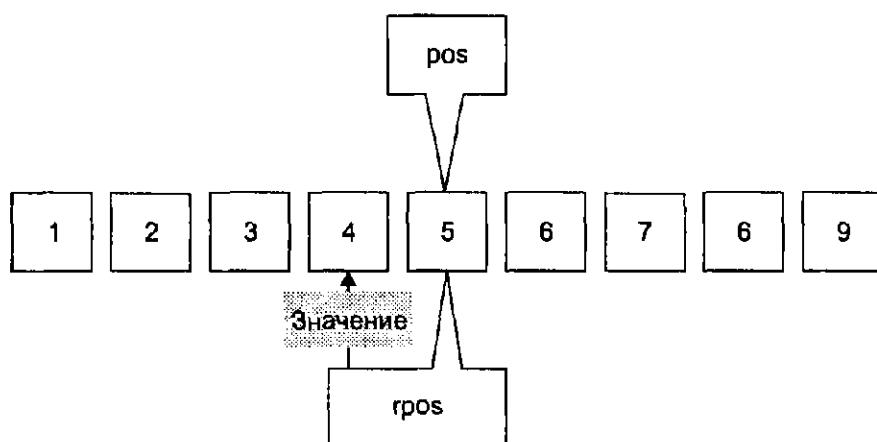


Рис. 7.4. Преобразование обычного итератора *pos* в обратный итератор *rpos*

Вам непонятен смысл такого решения? У него есть свои достоинства: например, вам ничего не придется делать при преобразовании интервала, заданного

двумя итераторами. Все элементы остаются действительными. Рассмотрим следующий пример:

```
// iter/reviter3.cpp
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;

void print (int elem)
{
    cout << elem << ' ';
}

int main()
{
    deque<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // Поиск позиции элемента со значением 2
    deque<int>::iterator pos1;
    pos1 = find (coll.begin(), coll.end(),      // Интервал
                 2);                           // Значение

    // Поиск позиции элемента со значением 7
    deque<int>::iterator pos2;
    pos2 = find (coll.begin(), coll.end(),      // Интервал
                 7);                           // Значение

    // Вывод всех элементов в интервале [pos1,pos2)
    for_each (pos1, pos2,      // Интервал
              print);           // Операция
    cout << endl;

    // Преобразование итераторов в обратные итераторы
    deque<int>::reverse_iterator rpos1(pos1);
    deque<int>::reverse_iterator rpos2(pos2);

    // Вывод всех элементов интервала [pos1,pos2) в обратном порядке
    for_each (rpos2, rpos1,      // Интервал
              print);           // Операция
    cout << endl;
}
```

Итераторы `pos1` и `pos2` определяют полуоткрытый интервал, в который входит элемент со значением 2, но не входит элемент со значением 7. Когда итера-

торы, определяющие этот интервал, преобразуются в обратные итераторы, интервал остается действительным и может быть обработан в обратном порядке. Результат выполнения программы выглядит так:

```
2 3 4 5 6
6 5 4 3 2
```

Итак:

- `rbegin()` — это просто:

```
контейнер::reverse_iterator(end())
```

- `rend()` — это просто:

```
контейнер::reverse_iterator(begin())
```

Конечно, константные итераторы преобразуются к типу `const_reverse_iterator`.

Преобразование обратных итераторов в обычные функцией `base`

Обратный итератор можно снова преобразовать в обычный. Для этой цели обратные итераторы поддерживают функцию `base()`:

```
namespace std {
    template <class Iterator>
    class reverse_iterator ... {

        ...
        Iterator base() const;
        ...
    };
}
```

Пример использования `base()`:

```
// iter/reviter4.cpp
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // Поиск позиции элемента со значением 5
    list<int>::iterator pos;
    pos = find (coll.begin(), coll.end(),      // Интервал
                5);                           // Значение
```

```
// Вывод значения элемента
cout << "pos: " << *pos << endl;

// Преобразование итератора в обратный итератор
list<int>::reverse_iterator rpos(pos);

// Вывод значения элемента, на который ссылается обратный итератор
cout << "rpos: " << *rpos << endl;

// Преобразование обратного итератора в обычный
list<int>::iterator rrpos;
rrpos = rpos.base();

// Вывод значения элемента, на который ссылается итератор
cout << "rrpos: " << *rrpos << endl;
}
```

Результат выполнения программы выглядит так:

```
pos: 5
rpos: 4
rrpos: 5
```

Преобразование `*rpos.base()` эквивалентно преобразованию обратного итератора. Иначе говоря, физическая позиция (элемент, на который ссылается итератор) сохраняется, а логическая позиция (значение элемента) перемещается. Другой пример использования функции `base()` приведен на с. 350.

Итераторы вставки

Итератор вставки представляет собой итераторный адаптер, преобразующий присваивание нового значения во вставку нового значения. С помощью итераторов вставки алгоритмы вставляют новые значения вместо того, чтобы записывать их на место старых. Все итераторы вставки относятся к категории итераторов вывода, то есть поддерживают только вставку новых значений (см. с. 259).

Функциональность итераторов вставки

Обычно алгоритмы присваивают значения элементам, на которые ссылаются приемные итераторы. Для примера рассмотрим алгоритм `copy()` (см. с. 358):

```
namespace std {
    template <class InputIterator, class OutputIterator>
    OutputIterator copy (InputIterator from_pos, // Начало источника
                        InputIterator from_end, // Конец источника
                        OutputIterator to_pos) // Начало приемника
    {
        while (from_pos != from_end) {
            *to_pos = *from_pos; // Копирование значений
            ++from_pos;
            ++to_pos;
        }
    }
}
```

```

        ++to_pos;
    }
    return to_pos;
}
}

```

Цикл продолжается до тех пор, пока итератор начала источника не дойдет до итератора конца источника. Внутри цикла значение, связанное с итератором `from_pos`, присваивается значению, связанному с итератором приемника `to_pos`, после чего оба итератора увеличиваются. В этом цикле особенно интересна команда присваивания нового значения:

`*to_pos = значение`

Итератор вставки преобразует эту команду присваивания в команду вставки. Но на самом деле в этой команде задействованы две операции: сначала оператор `*` возвращает элемент, на который ссылается итератор, после чего оператор `=` присваивает новое значение. В реализации итераторов вставки обычно используется особый «фокус», состоящий из двух этапов.

1. Оператор `*` реализуется как фиктивная операция, которая просто возвращает `*this`. Это означает, что для итераторов вставки выражение `*pos` эквивалентно `pos`.
2. Оператор присваивания реализуется так, что преобразуется в команду вставки. В конечном счете оператор вставки вызывает одну из функций контейнера `push_back()`, `push_front()` или `insert()`.

Это означает, что для итераторов вставки включение нового значения может выполняться командой `pos=значение` вместо `*pos=значение`. Впрочем, не стоит полагаться на подобные особенности реализации итераторов. Правильная команда присваивания должна иметь вид:

`*pos=значение`

Оператор `++` реализуется как фиктивная операция, которая просто возвращает `*this`. Следовательно, вы не можете изменить позицию итератора вставки. В табл. 7.7 перечислены все операции итераторов вставки.

Таблица 7.7. Операции итераторов вставки

Выражение	Описание
<code>*iter</code>	Фиктивная операция (возвращает <code>iter</code>)
<code>iter1 = value</code>	Вставка <code>value</code>
<code>++iter</code>	Фиктивная операция (возвращает <code>iter</code>)
<code>iter++</code>	Фиктивная операция (возвращает <code>iter</code>)

Разновидности итераторов вставки

Стандартная библиотека C++ поддерживает три разновидности итераторов вставки: конечные, начальные и общие. Они различаются в зависимости от позиции,

в которой вставляется новое значение, и вызывают разные функции своего контейнера. Отсюда следует, что итератор вставки всегда должен быть инициализирован своим контейнером.

Во всех разновидностях итераторов вставки определены вспомогательные функции для создания и инициализации. В табл. 7.8 перечислены разновидности итераторов вставки и их свойства.

Таблица 7.8. Разновидности итераторов вставки

Название	Класс	Вызываемая функция	Создание
Конечный итератор вставки	back_insert_iterator	push_back(value)	back_inserter(cont)
Начальный итератор вставки	front_insert_iterator	push_front (value)	front_inserter(cont)
Общий итератор вставки	insert_iterator	insert(pos,value)	inserter(cont,pos)

Естественно, контейнер должен поддерживать функции,ываемые итератором вставки; в противном случае использовать соответствующий итератор не удастся. Из-за этого конечные итераторы вставки поддерживаются только для векторов, деков, списков и строк, а начальные итераторы вставки — только для деков и списков. Ниже итераторы вставки рассматриваются более подробно.

Конечные итераторы вставки

Конечный итератор вставки присоединяет новое значение в конец контейнера вызовом функции `push_back()` (см. с. 247). Функция `push_back()` определена только в векторах, деках, списках и строках, поэтому конечные итераторы вставки поддерживаются только этими контейнерами стандартной библиотеки C++.

Конечный итератор вставки должен инициализироваться контейнером в момент создания. Для этого удобнее всего воспользоваться функцией `back_inserter()`. Ниже приведен пример применения конечного итератора вставки.

```
// iter/backins.cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    // Создание конечного итератора вставки для coll
    // - неудобный способ
    back_insert_iterator<vector<int> > iter(coll);

    // Вставка элементов через обычный интерфейс итераторов
    *iter = 1;
```

```

iter++;
*iter = 2;
iter++;
*iter = 3;
PRINT_ELEMENTS(coll);

// Создание конечного итератора вставки и вставка элементов
// - удобный способ
back_inserter(coll) = 44;
back_inserter(coll) = 55;
PRINT_ELEMENTS(coll);

// Присоединение всех элементов контейнера с использованием
// конечного итератора вставки
// - резервирование памяти для предотвращения ее перераспределения
coll.reserve(2*coll.size());
copy (coll.begin(), coll.end(), // Источник
      back_inserter(coll)); // Приемник
PRINT_ELEMENTS(coll);
}

```

Результат выполнения программы выглядит так:

```

1 2 3
1 2 3 44 55
1 2 3 44 55 1 2 3 44 55

```

Не забудьте зарезервировать достаточный объем памяти перед вызовом `copy()`. В противном случае добавление новых элементов конечным итератором вставки может привести к тому, что все остальные итераторы, относящиеся к вектору, станут недействительными. Другими словами, если не зарезервировать достаточно памяти, во время работы алгоритм может испортить переданные ему итераторы источника.

Строки также поддерживают интерфейс контейнеров STL, включая функцию `push_back()`. Соответственно конечные итераторы вставки могут использоваться для присоединения символов к строке. Пример приведен на с. 484.

Начальные итераторы вставки

Начальный итератор вставки вставляет новое значение в начало контейнера вызовом функции `push_front()` (см. с. 247). Функция `push_front()` определена только для деков и списков, поэтому начальные итераторы вставки поддерживаются только этими контейнерами стандартной библиотеки C++.

Начальный итератор вставки должен инициализироваться контейнером в момент создания. Для этого удобнее всего воспользоваться функцией `front_inserter()`. Ниже приведен пример использования начального итератора вставки.

```

// iter/frontins.cpp
#include <iostream>
#include <list>

```

```
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
    list<int> coll;

    // Создание начального итератора вставки для coll
    // - неудобный способ
    front_insert_iterator<list<int>> iter(coll);

    // Вставка элементов через обычный интерфейс итераторов
    *iter = 1;
    iter++;
    *iter = 2;
    iter++;
    *iter = 3;

    PRINT_ELEMENTS(coll);

    // Создание начального итератора вставки и вставка элементов
    // - удобный способ
    front_inserter(coll) = 44;
    front_inserter(coll) = 55;

    PRINT_ELEMENTS(coll);

    // Вставка всех элементов контейнера с использованием
    // начального итератора вставки
    copy (coll.begin(), coll.end(),      // Источник
          front_inserter(coll));        // Приемник

    PRINT_ELEMENTS(coll);
}
```

Результат выполнения программы выглядит так:

```
3 2 1
55 44 3 2 1
1 2 3 44 55 55 44 3 2 1
```

Обратите внимание: начальный итератор вставки добавляет элементы в обратном порядке. Это происходит из-за того, что следующий элемент всегда вставляется перед предыдущим.

Общие итераторы вставки

Общий итератор вставки инициализируется двумя значениями: контейнером и позицией вставки элементов. На основании этой информации вызывается функция `insert()`, которой заданная позиция передается в качестве аргумента.

Функция `inserter()` обеспечивает удобный способ создания и инициализации общего итератора вставки.

Общие итераторы вставки поддерживаются всеми стандартными контейнерами, потому что необходимая функция `insert()` определена во всех контейнерах. Тем не менее в ассоциативных контейнерах (множествах и отображениях) передаваемая позиция имеет рекомендательный, а не обязательный характер, поскольку правильная позиция элемента определяется его значением. За подробностями обращайтесь к описанию функции `insert()` на с. 245.

После вставки общий итератор вставки переходит в позицию вставленного элемента. Выполняются следующие команды:

```
pos = контейнер.insert(pos, value);
++pos;
```

Присваивание итератору возвращаемого значения функции `insert()` гарантирует, что позиция итератора всегда остается действительной. Без присваивания новой позиции в деках, векторах и строках общий итератор вставки сам делал бы себя недействительным, поскольку каждая операция вставки, по крайней мере теоретически, может сделать недействительными все итераторы, ссылающиеся на контейнер.

Пример использования общего итератора вставки:

```
// iter/inserter.cpp
#include <iostream>
#include <set>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
    set<int> coll;

    // Создание общего итератора вставки для coll
    // - неудобный способ
    insert_iterator<set<int> > iter(coll,coll.begin());

    // Вставка элементов через обычный интерфейс итераторов
    *iter = 1;
    iter++;
    *iter = 2;
    iter++;
    *iter = 3;

    PRINT_ELEMENTS(coll,"set: ");

    // Создание общего итератора вставки и вставка элементов
    // - удобный способ
    inserter(coll,coll.end()) = 44;
```

```
    inserter(coll, coll.end()) = 55;

    PRINT_ELEMENTS(coll, "set: ");

    // Вставка всех элементов контейнера с использованием
    // общего итератора вставки
    list<int> coll2;
    copy (coll.begin(), coll.end(),           // Источник
          inserter(coll2.begin()));           // Приемник

    PRINT_ELEMENTS(coll2, "list: ");

    // Вставка всех элементов контейнера перед вторым элементом
    // с использованием общего итератора вставки
    copy (coll.begin(), coll.end(),           // Источник
          inserter(coll2.++coll2.begin()));   // Приемник

    PRINT_ELEMENTS(coll2, "list: ");
}
```

Результат выполнения программы выглядит так:

```
set: 1 2 3
set: 1 2 3 44 55
set: 1 2 3
list: 1 2 3 44 55
list: 1 1 2 3 44 55 2 3 44 55
```

Вызовы `copy()` доказывают, что общий итератор вставки сохраняет порядок следования элементов. При втором вызове `copy()` в качестве аргумента передается позиция внутри интервала.

Пользовательские итераторы вставки для ассоциативных контейнеров

Как упоминалось выше, в ассоциативных контейнерах аргумент позиции, передаваемый общему итератору вставки, носит исключительно рекомендательный характер. Рекомендация может повысить скорость вставки, однако возможно и ее снижение. Например, если вставленные элементы хранятся в обратном порядке, то рекомендация лишь снизит эффективность программы — ведь поиск позиций вставки всегда начинается с неверной точки. Таким образом, плохая рекомендация хуже, чем отсутствие рекомендации. Это хороший пример ситуации, в которой уместно использовать расширение стандартной библиотеки C++. Пример такого расширения приведен на с. 291.

Потоковые итераторы

Потоковым итератором называется итераторный адаптер, использующий поток данных в качестве источника или приемника алгоритма. В частности, потоковый итератор ввода читает элементы из входного потока данных, а потоковый итератор вывода записывает данные в выходной поток данных.

Особую разновидность потоковых итераторов составляют *итераторы потоковых буферов*, предназначенные для чтения или записи в потоковый буфер. Итераторы потоковых буферов рассматриваются на с. 638.

Потоковые итераторы вывода

Потоковые итераторы вывода записывают присваиваемые значения в выходной поток данных. Это позволяет напрямую выводить результаты работы алгоритмов в потоки данных через стандартный интерфейс итераторов. Реализация потоковых итераторов вывода основана на тех же принципах, что и реализация итераторов вставки (см. с. 275). Единственное различие заключается в том, что присваивание нового значения преобразуется в операцию вывода оператором <<. В табл. 7.9 перечислены операции потоковых итераторов вывода.

Таблица 7.9. Операции потоковых итераторов вывода

Выражение	Описание
<code>ostream_iterator<T>(ostream)</code>	Создание потокового итератора вывода для потока данных <code>ostream</code>
<code>ostream_iterator<T>(ostream,delim)</code>	Создание потокового итератора вывода для потока данных <code>ostream</code> с разделением выводимых значений строкой <code>delim</code> (параметр <code>delim</code> относится к типу <code>const char*</code>)
<code>*iter</code>	Фиктивная операция (возвращает <code>iter</code>)
<code>iter1 = value</code>	Записывает <code>value</code> в поток данных <code>ostream</code> : <code>ostream<<value</code> (после чего выводится разделитель <code>delim</code> , если он задан)
<code>++iter</code>	Фиктивная операция (возвращает <code>iter</code>)
<code>iter++</code>	Фиктивная операция (возвращает <code>iter</code>)

При создании потокового итератора вывода необходимо указать выходной поток данных, в который должны записываться данные. В необязательном аргументе можно передать строку, которая должна выводиться между отдельными значениями. Без разделителя элементы будут выводиться непосредственно друг за другом.

Потоковые итераторы вывода определяются для типа элемента `T`:

```
namespace std {
    template <class T,
              class chart = char,
              class traits = char_traits<char_T> >
    class ostream_iterator;
```

Необязательные второй и третий аргументы шаблона определяют тип используемого потока данных (их смысл разъясняется на с. 562)¹.

¹ В устаревших системах необязательные параметры шаблона отсутствуют.

Следующий пример демонстрирует использование потоковых итераторов вывода.

```
// iter/ostriter.cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    // Создание потокового итератора вывода для потока cout
    // - выводимые значения разделяются символом новой строки
    ostream_iterator<int> intWriter(cout, "\n");

    // Запись элементов через обычный интерфейс итераторов
    *intWriter = 42;
    intWriter++;
    *intWriter = 77;
    intWriter++;
    *intWriter = -5;

    // Создание коллекции с элементами от 1 до 9
    vector<int> coll;
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // Запись всех элементов без разделителей
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout));
    cout << endl;

    // Запись всех элементов с разделителем " < "
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout, " < "));
    cout << endl;
}
```

Результат выполнения программы выглядит так:

```
42
77
-5
123456789
1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 <
```

Разделитель относится к типу `const char*`, поэтому при передаче объекта типа `string` необходимо преобразовать его кциальному типу вызовом функции `c_str()` (см. с. 495). Пример:

```
string delim;
...
ostream_iterator<int>(cout,delim.c_str());
```

Потоковые итераторы ввода

Потоковые итераторы ввода являются противоположностью потоковых итераторов вывода. Потоковый итератор ввода читает элементы из входного потока данных. Это позволяет алгоритмам читать свои исходные данные прямо из потока. Тем не менее потоковые итераторы ввода устроены несколько сложнее потоковых итераторов вывода (как обычно, чтение является более сложной операцией, чем запись).

В момент создания потоковый итератор ввода инициализируется входным потоком данных, из которого будут читаться данные. Далее стандартный интерфейс итераторов ввода (см. с. 258) используется для чтения элементов оператором `>>`. Тем не менее чтение может завершиться неудачей (при достижении конца файла или возникновении ошибки), а интервальному источнику алгоритма необходима «конечная позиция». Обе проблемы решаются при помощи *итератора конца потока данных*, создаваемого конструктором по умолчанию потоковых итераторов ввода. Если попытка чтения завершается неудачей, все потоковые итераторы ввода превращаются в итераторы конца потока данных. Следовательно, после каждой операции чтения потоковый итератор ввода необходимо сравнивать с итератором конца потока данных и убеждаться в том, что он остался действительным. В табл. 7.10 перечислены все операции потоковых итераторов ввода.

Таблица 7.10. Операции потоковых итераторов ввода

Выражение	Описание
<code>istream_iterator<T>()</code>	Создание итератора конца потока данных
<code>istream_iterator<T>(istream)</code>	Создание потокового итератора ввода для потока данных <code>istream</code> (с чтением первого значения)
<code>*iter</code>	Обращение к ранее прочитанному значению (читает первое значение, если оно не было прочитано конструктором)
<code>iter1->member</code>	Обращение к переменной или функции ранее прочитанного значения
<code>++iter</code>	Читает следующее значение и возвращает итератор для его позиции
<code>iter++</code>	Читает следующее значение, но возвращает итератор для предыдущего значения
<code>iter1==iter2</code>	Проверка <code>iter1</code> и <code>iter2</code> на равенство
<code>iter1!=iter2</code>	Проверка <code>iter1</code> и <code>iter2</code> на неравенство

Конструктор потокового итератора ввода открывает поток данных и обычно читает первый элемент, поскольку в противном случае он не смог бы вернуть

первый элемент при вызове оператора * после инициализации. Тем не менее реализация может отложить первое чтение до первого вызова оператора *. Это означает, что потоковый итератор ввода не следует определять задолго до того, когда он действительно понадобится.

Потоковые итераторы вывода определяются для типа элемента T:

```
namespace std {
    template <class T,
              class charT = char,
              class traits = char_traits<char_T>,
              class Distance = ptrdiff_t>
    class istream_iterator;
}
```

Необязательные второй и третий аргументы шаблона определяют тип используемого потока данных (их смысл разъясняется на с. 562). Необязательный четвертый аргумент шаблона определяет тип разности итераторов¹.

Два потоковых итератора ввода равны, если выполняется одно из следующих условий:

- оба являются итераторами конца потока данных, то есть дальнейшее чтение невозможно;
- оба итератора могут читать из одного потока данных.

Следующий пример демонстрирует использование потокового итератора ввода.

```
// iter/istrriter.cpp
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
    // Создание потокового итератора ввода для чтения целых чисел из cin
    istream_iterator<int> intReader(cin);

    // Создание итератора конца потока данных
    istream_iterator<int> intReaderEOF;

    /* Пока удаётся читать лексемы потоковым итератором ввода.
     * - дважды вывести прочитанное значение
     */
    while (intReader != intReaderEOF) {
        cout << "once: " << *intReader << endl;
        cout << "once again: " << *intReader << endl;
        ++intReader;
    }
}
```

¹ В устаревших системах, не поддерживающих значения по умолчанию для параметров шаблонов, четвертый аргумент является обязательным (и передается на месте второго аргумента), а аргументы типа потока данных отсутствуют.

Пусть входными данными программы является следующая последовательность:

```
1 2 3 f 4
```

Если запустить программу с этими входными данными, результат ее выполнения будет выглядеть так:

```
once:      1
once again: 1
once:      2
once again: 2
once:      3
once again: 3
```

Как видите, символ «f» во входном потоке данных завершает работу программы. Из-за ошибки формата дальнейшее чтение невозможно, поэтому потоковый итератор ввода `intReader` равен итератору конца потока данных `intReaderEOF` (а условие завершения цикла становится равным `false`).

Еще один пример использования потоковых итераторов

В следующем примере задействованы обе разновидности потоковых итераторов, а также функция `advance()`:

```
// iter/advance2.cpp
#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
     * - записывать каждую третью строку
     */
    while (cinPos != istream_iterator<string>()) {
        // Пропустить следующие две строки
        advance (cinPos, 2);

        // Чтение и запись третьей строки
        if (cinPos != istream_iterator<string>()) {
            *coutPos++ = *cinPos++;
        }
    }
    cout << endl;
}
```

Вспомогательная функция `advance()` перемещает итератор к другой позиции (см. с. 265). При использовании с потоковыми итераторами вывода она позво-

ляет пропускать лексемы во входном потоке данных. Например, рассмотрим следующие входные данные¹:

```
No one objects if you are doing  
a good programming job for  
someone whom you respect.
```

Для этих данных будет получен такой результат:

```
objects are good for you
```

Прежде чем обращаться к значению `*cinPos` после вызова `advance()`, не забудьте убедиться в том, что потоковый итератор вывода остался действительным. Вызов оператора `*` для итератора конца потока данных приводит к непредсказуемым последствиям.

Другие примеры использования потоковых итераторов при чтении из потока данных и записи в поток данных приведены на с. 118, 361 и 378.

Грактовка итераторов

Итераторы делятся на категории (см. с. 257) в соответствии со своими специфическими возможностями. Иногда бывает удобно и даже необходимо переопределить поведение итератора для других категорий. Такое переопределение выполняется при помощи механизма тегов и трактовок итераторов.

Для каждой категории итераторов в стандартной библиотеке C++ определяется *тег*, предназначенный для ссылок на данную категорию:

```
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 наследование. Так, из представленной иерархии следует вывод, что любой прямой итератор `forward_iterator_tag` является разновидностью итератора ввода `input_iterator_tag`. Но заметьте, что тег прямых итераторов

¹ Спасибо Эндрю Кенигу (Andrew Koenig) за хороший пример.

объявлен производным только от тега итераторов ввода, а не от тега итераторов вывода. Соответственно произвольный прямой итератор *не является* разновидностью итератора вывода. Для прямых итераторов определены особые требования, которые отличают их от итераторов вывода. За подробностями обращайтесь на с. 260.

При написании универсального кода важна не только категория итераторов. Например, вам может потребоваться тип элементов, на которые ссылается итератор. По этой причине в стандартной библиотеке C++ имеется специальный шаблон структуры, позволяющий определить варианты *трактовки итераторов* (iterator traits). Эта структура содержит наиболее существенную информацию об итераторе и предоставляет общий интерфейс ко всем определениям типов, ассоциированным с итератором (категория, тип элементов и т. д.):

```
namespace std {
    template <class T>
    struct iterator_traits {
        typedef typename T::value_type           value_type;
        typedef typename T::difference_type     difference_type;
        typedef typename T::iterator_category   iterator_category;
        typedef typename T::pointer             pointer;
        typedef typename T::reference          reference;
    };
}
```

В этом шаблоне *T* представляет тип итератора. Это означает, что в программе для произвольного итератора можно получить доступ к его категории, типу элементов и т. д. Например, следующее выражение определяет тип значений, передаваемых итератором типа *T*:

```
typename std::iterator_traits<T>::value_type
```

У такой структуры есть два важных достоинства:

- она гарантирует, что итератор предоставляет все необходимые определения типов;
- она может (в определенной степени) настраиваться для специальных итераторов.

Например, специализация применяется для обычных указателей, которые тоже могут использоваться в качестве итераторов:

```
namespace std {
    template <class 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;
    };
}
```

Любой тип «указателя на T» определяется как относящийся к категории итераторов произвольного доступа. Соответствующая частичная специализация существует и для константных указателей (`const T*`).

Написание унифицированных функций для итераторов

Благодаря механизму трактовки итераторов появляется возможность писать унифицированные функции, которые в зависимости от категории итератора используют производные определения типов или выбирают код реализации.

Использование типов итераторов

Простым примером использования механизма трактовки итераторов является алгоритм, которому необходима временная переменная для хранения элементов. Временная переменная просто объявляется в следующем виде:

```
typename std::iterator_traits<T>::value_type tmp;
```

Здесь `T` — тип итератора.

Другой пример — алгоритм циклического сдвига элементов:

```
template <class 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) {
        // Сохранить значение первого элемента
        value_type tmp(*beg);

        // Сдвиг следующих значений
        ...
    }
}
```

Выбор категории итератора

Выбор реализации в зависимости от категории итератора выполняется в два этапа.

1. При вызове другой функции ваша шаблонная функция передает ей категорию итератора в качестве дополнительного аргумента. Пример:

```
template <class Iterator>
inline void foo (Iterator beg, Iterator end)
{
    foo (beg, end,
          std::iterator_traits<Iterator>::iterator_category());
```

2. Другая функция реализуется для всех категорий итераторов, имеющих специальную реализацию, не унаследованную от одной из категорий итераторов.
Пример:

```
// foo() для двунаправленных итераторов
template <class BiIterator>
void foo (BiIterator beg, BiIterator end,
          std::bidirectional_iterator_tag)
{
    ...
}
```

Версия для итераторов произвольного доступа могла бы, например, использовать возможности, не поддерживаемые двунаправленными итераторами. Благодаря иерархической структуре тегов итераторов (см. с. 287) можно предоставить одну реализацию для нескольких категорий итераторов.

Выбор реализации для функции `distance`

В следующем примере демонстрируется описанная методика выбора реализации в зависимости от типа итератора. Мы рассмотрим реализацию вспомогательной функции `distance()`, возвращающей расстояние между двумя позициями итераторов и их элементов (см. с. 267). Реализация для итераторов произвольного доступа сводится к простому использованию оператора `-`. Для остальных категорий итераторов возвращается количество операций `++`, необходимых для достижения конца интервала.

```
// Унифицированная реализация функции distance()
template <class Iterator>
typename std::iterator_traits<Iterator>::difference_type
distance (Iterator pos1, Iterator pos2)
{
    return distance (pos1, pos2,
                      std::iterator_traits<Iterator>
                      ::iterator_category());
}

// Реализация функции distance() для итераторов произвольного доступа
template <class RaIterator>
typename std::iterator_traits<RaIterator>::difference_type
distance (RaIterator pos1, RaIterator pos2,
          std::random_access_iterator_tag)
{
    return pos2 - pos1;

}

// Реализация функции distance() для итераторов ввода.
// прямых и двунаправленных итераторов
template <class 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;
}

```

Тип разности итераторов используется как тип возвращаемого значения. Обратите внимание: во второй версии указан тег итераторов ввода, поэтому реализация автоматически используется прямыми и двунаправленными итераторами, теги которых являются производными от `input_iterator_tag`.

Пользовательские итераторы

Давайте напишем свой итератор. Как упоминалось в предыдущем разделе, для этого нужно предоставить варианты трактовки пользовательского итератора, что можно сделать двумя способами.

- Передача пяти необходимых типов в общей структуре `iterator_traits` (см. с. 288).
- Предоставление (частичной) специализации структуры `iterator_traits`.

Для использования первого способа в стандартную библиотеку C++ включен специальный базовый класс `iterator<>`, содержащий определения типов. Вам остается лишь передать нужные типы¹:

```

class MyIterator
    : public std::iterator<std::bidirectional_iterator_tag,
                           type, std::ptrdiff_t, type*, type&> {
    ...
};

```

Первый параметр шаблона определяет категорию итератора, второй — тип элемента, третий — тип разности, четвертый — тип указателя, а пятый — тип ссылки. Последние три аргумента не обязательны, по умолчанию им присваиваются значения `ptrdiff_t`, `type*` и `type&`. Во многих случаях достаточно следующего определения:

```

class MyIterator
    : public std::iterator<std::bidirectional_iterator_tag, type> {
    ...
};

```

¹ В прежних версиях STL вместо типа `iterator` использовались вспомогательные типы `input_iterator`, `output_iterator`, `forward_iterator`, `bidirectional_iterator` и `random_access_iterator`.

Ниже показано, как написать пользовательский итератор на примере итератора вставки для ассоциативных контейнеров. В отличие от итераторов вставки стандартной библиотеки C++ (см. с. 275) наш итератор не использует позицию вставки.

Реализация класса итератора выглядит так:

```
// iter/assointer.hpp
#include <iostream>

/* Шаблон итератора вставки для ассоциативных контейнеров
 */
template <class Container>
class asso_insert_iterator
    : public std::iterator <std::output_iterator_tag,
                           void, void, void>
{
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 - пустая операция, которая возвращает сам итератор
    asso_insert_iterator<Container>& operator* () {
        return *this;
    }

    // Увеличение - пустая операция, которая возвращает сам итератор
    asso_insert_iterator<Container>& operator++ () {
        return *this;
    }
    asso_insert_iterator<Container>& operator++ (int) {
        return *this;
    }
};

/* Вспомогательная функция для создания итератора вставки
 */
template <class 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`. Чтобы задать категорию итератора, в `iterator` передается первый аргумент шаблона `output_iterator_tag`. Итераторы вывода используются только для записи, поэтому, как и для всех итераторов вывода, в качестве типов элемента и разности указывается `void`¹.

В момент создания итератор сохраняет свой контейнер в переменной `container`. Все присваиваемые значения вставляются в контейнер функцией `insert()`. Операторы `*` и `++` реализуют фиктивные операции, которые просто возвращают сам итератор. При использовании стандартного интерфейса итераторов следующее выражение возвращает `*this`:

```
*pos = value
```

Операция присваивания преобразуется в вызов `insert(value)` для соответствующего контейнера.

Кроме класса итератора вставки мы также определяем вспомогательную функцию `assoinserter` для создания и инициализации итератора. Следующая программа добавляет в множество несколько элементов, используя пользовательский итератор вставки:

```
// iter/assointer.cpp

#include <iostream>
#include <set>
#include <algorithm>
using namespace std;

#include "print.hpp"

#include "assointer.hpp"

int main()
{
    set<int> coll;

    // Создание итератора вставки для coll
    // - неудобный способ
    asso_insert_iterator<set<int> > iter(coll);

    // Вставка элементов через обычный интерфейс итераторов
    *iter = 1;
    iter++;
}
```

¹ В прежних версиях STL класс `assointer` объявляется производным от класса `output_iterator` без параметров шаблона.

```
*iter = 2;
iter++;
*iter = 3;

PRINT_ELEMENTS(coll);

// Создание итератора вставки для coll и вставка элементов
// - удобный способ
asso_inserter(coll) = 44;
asso_inserter(coll) = 55;

PRINT_ELEMENTS(coll);

// Использование итератора вставки с алгоритмом
int vals[] = { 33, 67, -4, 13, 5, 2 };
copy (vals, vals+(sizeof(vals)/sizeof(vals[0])), // Источник
      asso_inserter(coll)); // Приемник

PRINT_ELEMENTS(coll);
}
```

Результат выполнения программы выглядит так:

```
1 2 3
1 2 3 44 55
-4 1 2 3 5 13 33 44 55 67
```

8 Объекты функций STL

Эта глава посвящена *объектам функций*, или *функторам*, упоминавшимся на с. 134. Рассматриваются все стандартные объекты функций и функциональные адаптеры, объясняется концепция функциональной композиции и принципы написания пользовательских объектов функций.

Концепция объектов функций

Объектом функции, или функтором, называется объект, для которого определен оператор `()`. Так, в следующем фрагменте выражение `fo()` означает не вызов функции `fo()`, а вызов оператора `()` объекта функции `fo`:

```
FunctionObjectType fo:  
...  
fo(...);
```

На первый взгляд кажется, что объект функции — это самая обычная функция, зачем-то записанная более сложным способом. В обычной функции все необходимые команды включаются в тело функции:

```
void fo() {  
    команды  
}
```

В случае объекта функции команды включаются в тело оператора `()` класса объекта функции:

```
class FunctionObjectType {  
public:  
    void operator() () {  
        команды  
    }  
};
```

Действительно, такое определение более сложно, однако у него есть три важных преимущества.

- Объект функции ведет себя более разумно, потому что может обладать состоянием. Например, можно создать два экземпляра одной функции, представленной объектом функции, которые могут одновременно обладать разными состояниями. Для обычных функций это невозможно.
- Каждый объект функции обладает и некоторым типом. Это означает, что вы можете передать объект функции шаблону, определяя какие-то аспекты его поведения, а контейнеры с разными объектами функций будут считаться относящимися к разным типам.
- Объект функции обычно работает быстрее указателя на функцию.

На с. 136 эти преимущества описаны более подробно, а пример на с. 137 показывает, что объект функции «умнее» обычной функции.

Далее приводятся два примера работы с объектами функций. В первом примере используется тот факт, что каждый объект функции обычно обладает собственным типом. Второй пример показывает, как задействовать состояние объекта функции, а также демонстрирует одно интересное свойство алгоритма `for_each()`.

Объект функции в качестве критерия сортировки

Программисты часто работают с сортированными коллекциями элементов, относящихся к определенному классу (например, с коллекцией объектов `Person`). Но предположим, сортировка объектов должна осуществляться не обычным оператором `<`, а по специальному критерию, оформленному в виде функции. В таких случаях на помощь приходят объекты функций. 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 функции-предиката
 * - оператор () сравнивает два объекта Person
 */
class PersonSortCriterion {
public:
```

```
bool operator() (const Person& p1, const Person& p2) const {
    /* Первый объект Person меньше второго.
     * - если фамилия в первом объекте меньше фамилии во втором объекте:
     * - или если фамилии равны, а имя в первом объекте меньше.
     */
    return p1.lastname()<p2.lastname() ||
        (!(p2.lastname()<p1.lastname()) &&
         p1.firstname()<p2.firstname());
}
};

int main()
{
    // Объявление типа множества со специальным критерием сортировки
    typedef set<Person,PersonSortCriterion> PersonSet;

    // Создание коллекции
    PersonSet coll;
    ...

    // Выполнение операций с элементами
    PersonSet::iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        ...
    }
    ...
}
```

Множество `coll` использует специальный критерий сортировки `PersonSortCriterion`, определенный в виде объекта функции. `PersonSortCriterion` определяет оператор `()` так, что он сравнивает два объекта `Person` по полю фамилии, а если они равны — по имени. Конструктор `coll` автоматически создает экземпляр класса `PersonSortCriterion`, чтобы элементы сортировались в соответствии с этим критерием.

Обратите внимание: критерий сортировки `PersonSortCriterion` является *типовым*. Следовательно, он может передаваться в аргументе шаблона множества. Если бы критерий был оформлен в виде обычной функции (как на с. 133), это было бы невозможно.

Все множества с данным критерием сортировки образуют отдельный тип (в данном примере он называется `PersonSet`). Они не могут использоваться совместно с множествами, имеющими «обычный» или другой пользовательский критерий сортировки (в том числе участвовать в операциях присваивания). Это означает, что никакая операция с множеством не приведет к нарушению автоматической сортировки; впрочем, можно написать объект функции, представляющий разные критерии сортировки с одним типом (см. следующий подраздел). За дополнительной информацией о множествах и их критериях сортировки обращайтесь на с. 186.

Объекты функций с внутренним состоянием

В следующем примере показано, как при помощи объекта функции имитировать функцию, обладающую несколькими состояниями одновременно.

```
// fo/general.cpp
#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

class IntSequence {
private:
    int value;
public:
    // Конструктор
    IntSequence (int initialValue)
        : value(initialValue) {
    }

    // "Вызов функции"
    int operator() () {
        return value++;
    }
};

int main()
{
    list<int> coll;

    // Вставка значений от 1 до 9
    generate_n (back_inserter(coll),      // Начало
                9,                      // Количество элементов
                IntSequence(1));        // Генератор значений

    PRINT_ELEMENTS(coll);

    // Замена элементов от второго до предпоследнего значениями,
    // начинающимися с 42
    generate (++coll.begin(),           // Начало
              --coll.end(),          // Конец
              IntSequence(42));      // Генератор значений

    PRINT_ELEMENTS(coll);
}
```

В данном примере объект функции генерирует последовательность целых чисел. При каждом вызове оператор () возвращает текущее значение счетчика

и увеличивает его. Начальное значение счетчика передается в аргументе конструктора.

Два таких объекта функции используются алгоритмами `generate()` и `generate_n()`, записывающими генерированные значения в коллекцию. Выражение `IntSequence(1)` в следующей команде создает объект функции, инициализированный значением 1:

```
generate_n (back_inserter(coll),  
            9,  
            IntSequence(1));
```

Алгоритм `generate(n)` использует его девять раз для записи элемента, поэтому объект генерирует значения от 1 до 9. Аналогично, выражение `IntSequence(42)` генерирует последовательность, начиная со значения 42. Алгоритм `generate()` заменяет элементы, начиная с `++coll.begin()` и заканчивая `--coll.end()`¹. Результат выполнения программы выглядит так:

```
1 2 3 4 5 6 7 8 9  
1 42 43 44 45 46 47 48 9
```

Используя другие версии оператора `()`, вы сможете легко строить более сложные последовательности.

Объекты функций передаются по значению, а не по ссылке. Следовательно, алгоритм не изменяет состояния объекта функции. Например, следующий код генерирует последовательность, начинающуюся со значения 1, дважды:

```
IntSequence seq(1);      // Серия целых чисел, начинающаяся с 1  
  
// Вставка последовательности, начинающейся с 1  
generate_n (back_inserter(coll), 9, seq);  
  
// Повторная вставка последовательности, начинающейся с 1  
generate_n (back_inserter(coll), 9, seq);
```

Передача объектов функций по значению, а не по ссылке, хороша тем, что позволяет передавать константы и временные выражения. В противном случае передача конструкции `IntSequence(1)` было бы невозможна.

Впрочем, у передачи объектов функций по значению есть и недостатки: такая передача не позволяет учесть изменения в состоянии объекта. Алгоритм может изменить состояние объекта функции, однако вам не удастся получить итоговое состояние и обработать его в программе, потому что алгоритм создает внутреннюю копию объекта функции. Но что делать, если вам все-таки необходимо получить «результат» от алгоритма?

Существуют два способа получения «результата» при использовании объектов функций алгоритмами:

- передача объекта функции по ссылке;
- использование возвращаемого значения алгоритма `for_each()` (см. с. 301).

¹ Возможно, выражения `++coll.begin()` и `--coll.end()` не будут работать с векторами. Эта неприятная проблема обсуждается на с. 264.

Чтобы передать объект функции по ссылке, достаточно уточнить вызов алгоритма так, чтобы тип объекта функции представлял собой ссылку¹. Пример:

```
// fo/genera2.cpp
#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

class IntSequence {
private:
    int value;
public:
    // Конструктор
    IntSequence (int initialValue)
        : value(initialValue) {
    }

    // "Вызов функции"
    int operator() () {
        return value++;
    }
};

int main()
{
    list<int> coll;
    IntSequence seq(1);    // Серия целых чисел, начинающаяся с 1

    // Вставка значений от 1 до 4
    // - передача объекта функции по ссылке,
    // чтобы при следующем вызове значения начинались с 5
    generate_n<back_insert_iterator<list<int>>(
        int, IntSequence&>(back_inserter(coll)),    // Начало
        4,          // Количество элементов
        seq);      // Генератор значений
    PRINT_ELEMENTS(coll);

    // Вставка значений от 42 до 45
    generate_n (back_inserter(coll),           // Начало
                4,                      // Количество элементов
                IntSequence(42));       // Генератор значений
    PRINT_ELEMENTS(coll);

    // Продолжение первой последовательности
    // - передача объекта функции по ссылке,
    // чтобы при следующем вызове значения снова начинались с 5
}
```

¹ Спасибо Филипу Кёстеру (Philip Küster), указавшему на эту возможность.

```

    generate_n (back_inserter(coll),      // Начало
                4,                      // Количество элементов
                seq);                  // Генератор значений
    PRINT_ELEMENTS(coll);

    // Снова продолжить первую последовательность
    generate_n (back_inserter(coll),      // Начало
                4,                      // Количество элементов
                seq);                  // Генератор значений
    PRINT_ELEMENTS(coll);
}

```

Результат выполнения программы выглядит так:

```

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 `generate_n()` объект функции `seq` передается по ссылке, для чего производится уточнение аргументов шаблона:

```

generate_n<back_insert_iterator<list<int> >,
            int, IntSequence&>(back_inserter(coll),    // Начало
                                  4,                  // Количество элементов
                                  seq);              // Генератор значений

```

В результате внутреннее значение `seq` изменяется после вызова, а второе использование `seq` при третьем вызове `generate_n()` продолжает серию из первого вызова. Но на этот раз `seq` передается по значению, а не по ссылке:

```

generate_n (back_inserter(coll),      // Начало
                4,                      // Количество элементов
                seq);                  // Генератор значений

```

Следовательно, вызов не изменяет внутреннего состояния `seq`, поэтому следующий вызов `generate_n()` снова продолжает серию, начиная с 5.

Возвращаемое значение алгоритма `for_each`

Благодаря алгоритму `for_each()` хлопоты с реализацией объекта функции на базе подсчета ссылок для получения его итогового состояния оказываются излишними. Алгоритм `for_each()` обладает уникальной особенностью — он возвращает свой объект функции (другие алгоритмы этого делать не могут). Это означает, что вы можете получить информацию о состоянии объекта функции, проверяя возвращаемое значение алгоритма `for_each()`.

Следующая программа хорошо поясняет, как использовать возвращаемое значение алгоритма `for_each()`. В ней вычисляется среднее арифметическое числовой последовательности.

```

// fo/foreach3.cpp
#include <iostream>

```

```

#include <vector>
#include <algorithm>
using namespace std;

// Объект функции для вычисления среднего арифметического
class MeanValue {
private:
    long num;      // Счетчик элементов
    long sum;      // Сумма всех значений элементов
public:
    // Конструктор
    MeanValue () : num(0), sum(0) {
    }

    // "Вызов функции"
    // - обработка очередного элемента последовательности
    void operator() (int elem) {
        num++;           // Увеличение счетчика
        sum += elem;     // Прибавление значения
    }

    // Возвращение среднего арифметического
    double value () {
        return static_cast<double>(sum) / static_cast<double>(num);
    }
};

int main()
{
    vector<int> coll;

    // Вставка элементов от 1 до 8
    for (int i=1; i<=8; ++i) {
        coll.push_back(i);
    }

    // Вычисление и вывод среднего арифметического
    MeanValue mv = for_each (coll.begin(), coll.end(), // Интервал
                           MeanValue());           // Операция
    cout << "mean value: " << mv.value() << endl;
}

```

Вызов `MeanValue()` создает объект функции, который подсчитывает количество элементов и вычисляет сумму их значений. Передача этого объекта при вызове `for_each()` обеспечивает его вызов для каждого элемента контейнера `coll`:

```

MeanValue mv = for_each (coll.begin(), coll.end(),
                        MeanValue());

```

Объект функции, возвращенный алгоритмом `for_each()`, присваивается `mv`, поэтому после вызова можно запросить информацию о его состоянии в виде `mv.value()`. В итоге программа выводит следующий результат:

```
mean value: 4.5
```

Класс `MeanValue` можно дополнительно усовершенствовать, определив автоматическое преобразование к типу `double`. В этом случае среднее арифметическое, вычисленное `for_each()`, можно будет напрямую использовать в программе. Пример такого рода приведен на с. 335.

Предикаты и объекты функций

Предикатом называется функция или объект функции, возвращающий логическое значение (или значение, преобразуемое к `bool`). Однако не каждая функция, возвращающая логическую величину, является предикатом по правилам STL. Иногда это приводит к весьма странным последствиям. Рассмотрим следующий пример:

```
// fo.removeif.cpp
#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

class Nth {           // Объект функции возвращает true для n-го вызова
private:
    int nth;          // Номер вызова, для которого следует вернуть true
    int count;         // Счетчик вызовов
public:
    Nth (int n) : nth(n), count(0) {
    }
    bool operator() (int) {
        return ++count == nth;
    }
};

int main()
{
    list<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }
    PRINT_ELEMENTS(coll, "coll:      ");

    // Удаление третьего элемента
    list<int>::iterator pos;
```

```

pos = remove_if(coll.begin(), coll.end(), // Интервал
                 Nth(3)); // Критерий удаления
coll.erase(pos, coll.end());

PRINT_ELEMENTS(coll, "nth removed: ");
}

```

Программа определяет объект функции `Nth`, который возвращает `true` для каждого *n*-го вызова. Но если передать этот объект алгоритму `remove_if()`, выполняющему удаление всех элементов, для которых унарный предикат возвращает `true` (см. с. 371), вас ждет сюрприз:

```

coll:      1 2 3 4 5 6 7 8 9
nth removed: 1 2 4 5 7 8 9

```

Из коллекции удаляются два элемента, третий и шестой. Дело в том, что обычная реализация алгоритма создает внутреннюю копию предиката:

```

template <class ForwIter, class Predicate>
ForwIter std::remove_if(ForwIter beg, ForwIter end,
                       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);
    }
}

```

Для поиска первого удаляемого элемента алгоритм использует `find_if()`, но затем для обработки оставшихся элементов используется копия переданного предиката `op`. Объект функции `Nth` возвращается к прежнему состоянию и удаляет третий элемент из оставшихся (то есть шестой элемент коллекции).

Такое поведение не является ошибкой. Стандарт не ограничивает внутреннее копирование предиката в алгоритме. Таким образом, для надежной работы алгоритмов стандартной библиотеки C++ не следует передавать объект функции, поведение которого зависит от частоты его копирования или вызова. Иначе говоря, если унарный предикат вызывается с одинаковыми аргументами, то он всегда должен возвращать один и тот же результат. Вызов не должен изменять внутреннее состояние предиката, а копия предиката должна иметь точно такое же состояние, как оригинал. Чтобы состояние предиката не изменялось при вызове функции, оператор () рекомендуется объявлять в виде константной функции.

В принципе это странное поведение можно обойти и обеспечить нормальную работу алгоритма даже с такими объектами функций, как `Nth`, без сниже-

ния быстродействия. Для этого достаточно исключить из реализации `remove_if()` вызов `find_if()`:

```
template <class ForwIter, class 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);
    }
}
```

Вероятно, реализацию `remove_if()` стоило бы изменить (или подать заявку на ее изменение автору библиотеки). Несколько мне известно, в текущих реализациях эта проблема возникает только с алгоритмом `remove_if()`. Если использовать алгоритм `remove_copy_if()`, все работает нормально¹. Но если вы хотите, чтобы программа была действительно переосимой, никогда не полагайтесь на особенности реализации. Всегда объявляйте оператор вызова функции и предикаты как константные функции классов.

Стандартные объекты функций

Как упоминалось на с. 140, стандартная библиотека C++ содержит ряд стандартных объектов функций. Эти объекты перечислены в табл. 8.1.

Таблица 8.1. Стандартные объекты функций

Выражение	Описание
<code>negate<type>()</code>	$- \text{param}$
<code>plus<type>()</code>	$\text{param1} + \text{param2}$
<code>minus<type>()</code>	$\text{param1} - \text{param2}$
<code>multiplies<type>()</code> ²	$\text{param1} * \text{param2}$

продолжение ↗

¹ В настоящее время обсуждается вопрос о том, должна ли стандартная библиотека C++ гарантировать ожидаемое поведение в подобных случаях.

² В предыдущих версиях STL объект функции, выполнявший умножение, назывался `times`. Переименование произошло из-за конфликта имен с функциями стандартов операционных систем (X/Open, POSIX), а также потому, что идентификатор `multiplies` понятней.

Таблица 8.1 (продолжение)

Выражение	Описание
<code>divides<type>()</code>	<code>param1/param2</code>
<code>modulus<type>()</code>	<code>param1%param2</code>
<code>equal_to<type>()</code>	<code>param1==param2</code>
<code>not_equal_to<type>()</code>	<code>param1!=param2</code>
<code>less<type>()</code>	<code>param1<param2</code>
<code>greater<type>()</code>	<code>param1>param2</code>
<code>less_equal<type>()</code>	<code>param1<=param2</code>
<code>greater_equal<type>()</code>	<code>param1>=param2</code>
<code>logical_not<type>()</code>	<code>!param</code>
<code>logical_and<type>()</code>	<code>param1&&param2</code>
<code>logical_or<type>()</code>	<code>param1 param2</code>

Объект функции `less<>` является критерием по умолчанию при сортировке или сравнении объектов, поэтому он используется достаточно часто. По умолчанию сортировка всегда проводится по возрастанию (*элемент <следЭлемент*).

Чтобы использовать стандартные объекты функций, необходимо включить в программу заголовочный файл `<functional>`¹:

```
#include <functional>
```

В стандартную библиотеку C++ также входит специальный объект функции, который может использоваться при контекстной сортировке строк. За подробностями обращайтесь на с. 676.

Функциональные адаптеры

Функциональным адаптером называется объект, который позволяет комбинировать объекты функций друг с другом, с определенными значениями или со специальными функциями. Функциональные адаптеры тоже объявляются в заголовочном файле `<functional>`. Например, в следующей команде выражение `bind2nd(greater<int>(),42)` создает комбинированный объект функции для проверки условия «целое число больше 42»:

```
find_if (coll.begin(),coll.end(),           // Интервал
         bind2nd(greater<int>(),42))      // Критерий
```

Фактически `bind2nd()` преобразует бинарный объект функции (например, `greater<>`) в унарный объект функции. Второй параметр всегда используется в качестве второго аргумента бинарной функции, передаваемой в первом параметре. Следовательно, в приведенном примере `greater<>` всегда вызывается со

¹ В исходной версии STL заголовочный файл объектов функций назывался `<function.h>`.

вторым аргументом, равным 42. На с. 140 имеются и другие примеры использования функциональных адаптеров.

В табл. 8.2 перечислены классы стандартных функциональных адаптеров, входящих в стандартную библиотеку C++.

Таблица 8.2. Стандартные функциональные адаптеры

Выражение	Описание
bind1st(op,value)	op(value,param)
bind2nd(op,value)	op(value,param)
not1(op)	!op(param)
not2(op)	!op(param1,param2)

Функциональные адаптеры сами по себе являются объектами функций, поэтому их можно объединять с другими адаптерами и объектами функций для построения более мощных (и более сложных) выражений. Например, следующая команда находит первый четный элемент коллекции:

```
pos = find_if (coll.begin(), coll.end(),           // Интервал
                not1(bind2nd(modulus<int>(),2))); // Критерий
```

В этой команде выражение bind2nd(modulus<int>(),2) возвращает 1 для нечетных значений. Следовательно, оно может использоваться в качестве критерия для нахождения первого нечетного элемента, потому что значение 1 эквивалентно `true`. Адаптер `not1()` производит логическую инверсию результата, поэтому вся команда ищет первый элемент с четным значением.

Объединяя объекты функций при помощи функциональных адаптеров, можно строить достаточно сложные выражения. Подобный стиль программирования называется функциональной композицией. Тем не менее в стандартной библиотеке C++ отсутствуют некоторые адаптеры, которые были бы весьма удобны при композиции. Например, отсутствуют адаптеры для связывания двух предикатов по условиям «и»/«или» (например, «больше 4 и меньше 7»). Расширения стандартных функциональных адаптеров делают механизм композиции гораздо более мощным. Примеры таких расширений приведены на с. 313.

Функциональные адаптеры для функций классов

В стандартную библиотеку C++ включены дополнительные адаптеры, позволяющие вызвать некоторую функцию класса для каждого элемента коллекции (табл. 8.3).

Таблица 8.3. Функциональные адаптеры для функций классов

Выражение	Описание
mem_fun_ref(op)	Вызов op() как функции объекта
mem_fun (op)	Вызов op() как функции указателя на объект

В следующем примере адаптер `mem_fun_ref` вызывает функцию `print()` каждого элемента вектора:

```
// fo/memfun1a.cpp
class Person {
private:
    std::string name;
public:
    ...
    void print () const {
        std::cout << name << std::endl;
    }
    void printWithPrefix (std::string prefix) const {
        std::cout << prefix << name << std::endl;
    }
};

void foo (const std::vector<Person>& coll)
{
    using std::for_each;
    using std::bind2nd;
    using std::mem_fun_ref;

    // Вызов функции print() для каждого элемента вектора
    for_each (coll.begin(), coll.end(),
              mem_fun_ref(&Person::print));

    // Вызов функции printWithPrefix() для каждого элемента
    // - строка "person: " передается при вызове
    for_each (coll.begin(), coll.end(),
              bind2nd(mem_fun_ref(&Person::printWithPrefix),
                      "person: "));
}

int main()
{
    std::vector<Person> coll(5);
    foo(coll);

    std::vector<Person*> coll2;
    coll2.push_back(new Person);
    ptrfoo(coll2);
}
```

В функции `foo()` для каждого элемента вектора `coll` вызываются две функции класса `Person`:

- функция `Person::print()` вызывается без аргументов;
- функции `printWithPrefix()` при вызове передается аргумент.

Чтобы вызвать функцию `Person::print()`, мы передаем объект функции `mem_fun_ref(&Person::print)` алгоритму `for_each()`:

```
for_each (coll.begin(), coll.end(),
          mem_fun_ref(&Person::print));
```

Адаптер `mem_fun_ref` трансформирует обращение к элементу в вызов указанной функции класса.

Почему необходимо использовать адаптер? Потому что алгоритму нельзя напрямую передать функцию класса. Если попытаться это сделать, произойдет ошибка компиляции:

```
for_each (coll.begin(), coll.end(),
          &Person::print);      // ОШИБКА: невозможно вызвать оператор ()  
                      // для указателя на функцию класса
```

Проблема заключается в том, что `for_each()` вызывает оператор `()` для указателя, переданного в третьем аргументе, вместо функции класса, на которую он ссылается. Адаптер `mem_fun_ref` решает эту проблему, преобразуя вызов оператора `()`.

Как показывает второй вызов `for_each()`, адаптер `bind2nd` также позволяет передать один аргумент вызванной функции класса¹:

```
for_each (coll.begin(), coll.end(),
          bind2nd(mem_fun_ref(&Person::printWithPrefix),
                  "person: "));
```

Наверное, вас удивляет, что адаптер называется `mem_fun_ref`, а не просто `mem_fun`. Исторически сложилось так, что первой появилась другая версия функциональных адаптеров, которой и было присвоено название `mem_fun`. Адаптеры `mem_fun` предназначались для последовательностей, содержащих *указатели* на элементы. Возможно, во избежание путаницы их следовало бы назвать `mem_fun_ptr`. Следовательно, функции классов могут вызываться не только для серий объектов, но и для серий указателей на объекты. Пример:

```
// fo/memfun1b.cpp
void ptrfoo (const std::vector<Person*>& coll)
              // ^^^ указатель!
{
    using std::for_each;
    using std::bind2nd;
    using std::mem_fun;

    // Вызов функции print() для каждого объекта.
    // на который ссылается указатель
    for_each (coll.begin(), coll.end(),
              mem_fun(&Person::print));
```

¹ В прежних версиях STL и стандартной библиотеки C++ адаптеры для функций с одним аргументом назывались `mem_fun1` и `mem_fun1_ref` вместо `mem_fun` и `mem_fun_ref`.

```

// Вызов функции printWithPrefix () для каждого объекта,
// на который ссылается указатель
// - строка "person: " передается при вызове
for_each (coll.begin(), coll.end(),
           bind2nd(mem_fun(&Person::printWithPrefix),
                  "person: "));
}

```

Адаптеры `mem_fun_ref` и `mem_fun` позволяют вызывать функции классов без аргументов или с одним аргументом. Вызвать функцию с двумя аргументами подобным образом не удастся. Дело в том, что для реализации этих адаптеров необходимы вспомогательные объекты функций, предоставляемые для каждого вида функций. Например, вспомогательные классы для `mem_fun` и `mem_fun_ref` называются `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`.

Учтите, что функции классов, вызываемые `mem_fun_ref` и `mem_fun`, должны быть *константными*. К сожалению, стандартная библиотека C++ не предоставляет функциональных адаптеров для неконстантных функций классов (автор обнаружил это в ходе работы над книгой). Вероятно, это объясняется простым недосмотром (просто никто не знал, что это невозможно), и проблема может быть решена без особых хлопот. Будем надеяться на ликвидацию этого недостатка в будущих реализациях (и версиях стандарта).

Функциональные адаптеры для обычных функций

Функциональный адаптер `ptr_fun` позволяет использовать обычные функции с другими функциональными адаптерами (табл. 8.4).

Таблица 8.4. Функциональные адаптеры для обычных функций

Выражение	Описание
<code>ptr_fun(op)</code>	<code>*op(param)</code>
	<code>*op(param1,param2)</code>

Предположим, имеется глобальная функция, которая проверяет некоторое условие для передаваемого параметра:

```
bool check(int elem);
```

Поиск первого элемента, для которого проверка завершается неудачей, производится следующей командой:

```
pos = find_if (coll.begin(), coll.end(),
               not1(ptr_fun(check))); // Интервал
                           // Критерий поиска
```

Применить конструкцию `not1(check)` нельзя, поскольку `not1()` использует специальные типы, предоставляемые объектами функций (см. далее).

Вторая форма используется в ситуации, когда глобальная функция вызывается с двумя параметрами, а вам, например, она нужна как унарная функция:

```
// Поиск первой непустой строки
pos = find_if (coll.begin(), coll.end(),           // Интервал
                bind2nd(ptr_fun(strcmp), ""));    // Критерий поиска
```

В этом фрагменте функция `strcmp()` языка С сравнивает каждый элемент с пустой строкой языка С. Если строки совпадают, функция `strcmp()` возвращает 0 (эквивалент `false`). Таким образом, данный вызов `find_if()` возвращает позицию первого элемента, отличного от пустой строки. Другой пример использования функционального адаптера `ptr_fun` приведен на с. 318.

Написание пользовательских объектов функций для функциональных адаптеров

Вы можете написать собственный объект функции, однако, чтобы он работал с функциональными адаптерами, в этом объекте должны быть определены типы для аргументов и результата. Для этого в стандартную библиотеку C++ были включены специальные структуры:

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg    argument_type;
    typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1   first_argument_type;
    typedef Arg2   second_argument_type;
    typedef Result result_type;
};
```

Объявляя свой объект функции производным от одного из этих типов, вы легко обеспечиваете соблюдение этих требований и объект функции становится «совместимым с адаптерами».

В следующем примере приведено полное определение объекта функции для возведения первого аргумента в степень, заданную вторым аргументом:

```
// fo/fopow.hpp
#include <functional>
#include <cmath>

template <class T1, class T2>
struct fopow : public std::binary_function<T1, T2, T1>
{
    T1 operator() (T1 base, T2 exp) const {
        return std::pow(base,exp);
    }
};
```

Первый аргумент и возвращаемое значение относятся к одному типу T_1 , а показатель степени может относиться к другому типу T_2 . Передавая эту информацию `binary_function`, мы обеспечиваем определение необходимых типов. Тем не менее вместо использования `binary_function` типы можно определить напрямую. Как обычно бывает в STL, функциональные адаптеры соответствуют концепции чистой абстракции: любой объект, который *ведет себя* как объект функции, совместимый с функциональными адаптерами, *является* таковым.

Следующая программа показывает, как работать с пользовательским объектом функции `fopow`. В частности, продемонстрировано использование `fopow` с адаптерами `bind1st` и `bind2nd`:

```
// fo/fopow1.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Включение пользовательского объекта функции fopow<>
#include "fopow.hpp"

int main()
{
    vector<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // Вывод числа 3, возведенного в степень каждого элемента
    transform (coll.begin(), coll.end(),           // Источник
               ostream_iterator<int>(cout, " "), // Приемник
               bind1st(fopow<float,int>().3)); // Операция
    cout << endl;

    // Вывод всех элементов, возведенных в степень 3
    transform (coll.begin(), coll.end(),           // Источник
               ostream_iterator<int>(cout, " "), // Приемник
               bind2nd(fopow<float,int>().3)); // Операция
    cout << endl;
}
```

Программа выводит следующий результат:

```
3 9 27 81 243 729 2187 6561 19683
1 8 27 64 125 216 343 512 729
```

Обратите внимание: объект функции `fopow` реализован для типов `float` и `int`. Если использовать `int` как для основания, так и для показателя степени, вы сможете вызвать `pow()` с двумя аргументами типа `int`, но при этом будет нарушена

переносимость кода, поскольку в соответствии со стандартом функция `pow()` перегружается более чем для одного, но не для всех базовых типов:

```
transform (coll.begin(), coll.end(),
          ostream_iterator<int>(cout, " "),
          bind1st(fopow<int,int>(),3)); // ОШИБКА: неоднозначность
```

Более подробное описание этой проблемы приводится на с. 557.

Дополнительные композиционные адаптеры

Возможность объединения объектов функций играет важную роль в построении программного кода из готовых компонентов. Композиция позволяет конструировать очень сложные объекты функций из более простых частей. Конечно, было бы очень полезно, чтобы практически любое функциональное отношение представлялось в виде комбинации объектов функций. К сожалению, набор адаптеров в стандартной библиотеке C++ недостаточно широк. Например, не существует адаптера, который бы позволял объединить две унарные операции для формулировки критерия вида «одно и другое».

Теоретически в библиотеке пригодились бы следующие композиционные адаптеры.

- **f(g(elem))**. Общая форма унарной композиции — вложенные вызовы унарных предикатов, при которых результаты вызова предиката `g()` являются входными данными для предиката `f()`. Все выражение работает как унарный предикат.
- **f(g(elem1, elem2))**. Два элемента `elem1` и `elem2` передаются в аргументах бинарного предиката `g()`, а результаты, как и в предыдущем случае, являются входными данными для унарного предиката `f()`. Все выражение работает как бинарный предикат.
- **f(g(elem), h(elem))**. Элемент `elem` передается двум разным унарным предикатам `g()` и `h()`, а результаты обрабатываются бинарным предикатом `f()`. Все выражение работает как унарный предикат.
- **f(g(elem1), h(elem2))**. В этой форме два элемента `elem1` и `elem2` передаются двум разным унарным предикатам `g()` и `h()`, а результаты обрабатываются бинарным предикатом `f()`. Все выражение работает как бинарный предикат.

К сожалению, эти композиционные адаптеры не были стандартизированы, поэтому для них не существуют стандартных имен. В реализации STL от SGI были определены имена для двух из перечисленных адаптеров, однако сообщество программистов еще не выбрало наиболее подходящие. В табл. 8.5 приведены имена, которые мы будем использовать в этой книге.

В архиве библиотек C++ Boost (<http://www.boost.org>) можно найти как имена, которые должны использоваться в будущем, так и их полную реализацию. Ниже описаны три адаптера, которые часто встречаются в книге.

Таблица 8.5. Возможные имена композиционных адаптеров

Функциональность	В книге	В реализации STL от SGI
f(g(elem))	compose_f_gx	compose1
f(g(elem1,elem2))	compose_f_gxy	
f(g(elem),h(elem))	compose_f_gx_hx	compose2
f(g(elem1),h(elem2))	compose_f_gx_hy	

Унарные композиционные адаптеры

Унарные композиционные адаптеры являются весьма важными. Они также входят в реализацию STL от SGI.

Вложенные вычисления с использованием адаптера `compose_f_gx`

В простейшем композиционном адаптере результаты выполнения одной унарной операции становятся входными данными для другой унарной операции. Иначе говоря, этот адаптер просто обеспечивает вложенный вызов двух унарных объектов функций. Он понадобится для формулировки условий типа «прибавить 10 и умножить на 4».

Возможная реализация адаптера `compose_f_gx` (в реализации SGI используется имя `compose1`) выглядит так:

```
// fo/compose1.hpp
#include <functional>

/* Класс композиционного адаптера compose_f_gx
 */
template <class OP1, class OP2>
class compose_f_gx_t
: public std::unary_function<typename OP2::argument_type,
                           typename OP1::result_type>
{
private:
    OP1 op1; // Вычисление: op1(op2(x))
    OP2 op2;
public:
    // Конструктор
    compose_f_gx_t(const OP1& o1, const OP2& o2)
        : op1(o1), op2(o2) {
    }

    // Вызов функции
    typename OP1::result_type
    operator()(const typename OP2::argument_type& x) const {
        return op1(op2(x));
    }
};
```

```
/* Вспомогательные функции для адаптера compose_f_gx
 */
template <class OP1, class OP2>
inline compose_f_gx_t<OP1,OP2>
compose_f_gx (const OP1& o1, const OP2& o2) {
    return compose_f_gx_t<OP1,OP2>(o1,o2);
}
```

Пример использования адаптера compose_f_gx:

```
// fo/compose1.cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <iterator>
#include "print.hpp"
#include "compose1.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }
    PRINT_ELEMENTS(coll);

    // Для каждого элемента прибавить 10 и умножить на 5
    transform (coll.begin(),coll.end(),
               ostream_iterator<int>(cout," "),
               compose_f_gx(bind2nd(multiplies<int>(),5),
                           bind2nd(plus<int>(),10)));
    cout << endl;
}
```

Обратите внимание: сначала выполняется вторая операция, переданная compose_f_gx. Таким образом, следующая конструкция создает унарный объект функции, который сначала прибавляет 10, а потом умножает результат на 5:

```
compose_f_gx(bind2nd(multiplies<int>(),5),
            bind2nd(plus<int>(),10));
```

Результат выполнения программы:

```
1 2 3 4 5 6 7 8 9
55 60 65 70 75 80 85 90 95
```

Объединение двух критериев с использованием адаптера `compose_f_gx_hx`

Вероятно, из вспомогательных функциональных адаптеров самым важным является тот, который формирует единый критерий логическим объединением двух критериев. Он используется для формулировки условий типа «больше 4 и меньше 7».

Возможная реализация адаптера `compose_f_gx_hx` (в реализации SGI используется имя `compose2`) выглядит так:

```
// fo/compose21.hpp
#include <functional>

/* Класс композитного адаптера compose_f_gx_hx
 */
template <class OP1, class OP2, class OP3>
class compose_f_gx_hx_t
    : public std::unary_function<typename OP2::argument_type,
                                typename OP1::result_type>
{
private:
    OP1 op1; // Вычисление: op1(op2(x),op3(x))
    OP2 op2;
    OP3 op3;
public:
    // Конструктор
    compose_f_gx_hx_t (const OP1& o1, const OP2& o2, const OP3& o3)
        : op1(o1), op2(o2), op3(o3) {
    }

    // Вызов функции
    typename OP1::result_type
    operator()(const typename OP2::argument_type& x) const {
        return op1(op2(x),op3(x));
    }
};

/* Вспомогательные функции для адаптера compose_f_gx_hx
 */
template <class OP1, class OP2, class OP3>
inline compose_f_gx_hx_t<OP1,OP2,OP3>
compose_f_gx_hx (const OP1& o1, const OP2& o2, const OP3& o3) {
    return compose_f_gx_hx_t<OP1,OP2,OP3>(o1,o2,o3);
}
```

Адаптер `compose_f_gx_hx` использует первую операцию для объединения результатов двух унарных операций с одним объектом. Следующее выражение создает унарный предикат:

```
compose_f_gx_hx(op1,op2,op3)
```

Этот предикат вычисляет для каждого значения x :

```
op1(op2(x).op3(x))
```

Пример использования адаптера **compose_f_gx_hx**:

```
// fo/compose2.cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include "print.hpp"
#include "compose21.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    // Вставка элементов со значениями от 1 до 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }
    PRINT_ELEMENTS(coll);

    // Удаление всех элементов, больших 4, но меньших 7
    // - retain new end
    vector<int>::iterator pos;
    pos = remove_if (coll.begin(),coll.end(),
                      compose_f_gx_hx(logical_and<bool>(),
                                      bind2nd(greater<int>(),4),
                                      bind2nd(less<int>(),7)));

    // Стирание "удаленных" элементов из коллекции
    coll.erase(pos,coll.end());

    PRINT_ELEMENTS(coll);
}
```

Следующее выражение формирует унарный предикат для проверки условия «значение больше 4 и меньше 7»:

```
compose_f_gx_hx(logical_and<bool>(),
                  bind2nd(greater<int>(),4),
                  bind2nd(less<int>(),7));
```

Результат выполнения программы выглядит так:

```
1 2 3 4 5 6 7 8 9
1 2 3 4 7 8 9
```

Бинарные композиционные адаптеры

Один из бинарных композиционных адаптеров обрабатывает результат двух унарных операций, которым передаются разные элементы. Автор выбрал для этого адаптера имя `compose_f_gx_hy`. Возможная реализация выглядит так:

```
// fo/compose22.hpp
#include <functional>

/* Класс композиционного адаптера compose_f_gx_hy
 */
template <class OP1, class OP2, class OP3>
class compose_f_gx_hy_t
    : public std::binary_function<typename OP2::argument_type,
                                typename OP3::argument_type,
                                typename OP1::result_type>
{
private:
    OP1 op1; // Вычисление: op1(op2(x),op3(y))
    OP2 op2;
    OP3 op3;
public:
    // Конструктор
    compose_f_gx_hy_t (const OP1& o1, const OP2& o2, const OP3& o3)
        : op1(o1), op2(o2), op3(o3) {
    }

    // Вызов функции
    typename OP1::result_type
    operator()(const typename OP2::argument_type& x,
                const typename OP3::argument_type& y) const {
        return op1(op2(x),op3(y));
    }
};

/* Вспомогательная функция для композиционного адаптера compose_f_gx_hy
 */
template <class OP1, class OP2, class OP3>
inline compose_f_gx_hy_t<OP1,OP2,OP3>
compose_f_gx_hy (const OP1& o1, const OP2& o2, const OP3& o3) {
    return compose_f_gx_hy_t<OP1,OP2,OP3>(o1,o2,o3);
}
```

В следующем примере демонстрируется применение адаптера `compose_f_gx_hy`. В нем производится поиск подстроки без учета регистра символов:

```
// fo/compose3.cpp
#include <iostream>
#include <algorithm>
#include <functional>
```

```
#include <string>
#include <cctype>
#include "compose22.hpp"
using namespace std;

int main()
{
    string s("Internationalization");
    string sub("Nation");

    // Поиск подстроки без учета регистра символов
    string::iterator pos;                                // Стока
    pos = search (s.begin(),s.end(),                // Искомая подстрока
                  sub.begin(),sub.end(),
                  compose_f_gx_hy(equal_to<int>(),
                                     ptr_fun(::toupper),
                                     ptr_fun(::toupper)));
}

}

if (pos != s.end()) {
    cout << "\"" << sub << "\"" is part of "\"" << s << "\""
        << endl;
}
```

Программа выводит следующий результат:

"Nation" is part of "Internationalization"

На с. 482 приведен пример поиска подстроки без учета регистра символов, в котором адаптер `compose_f_gx_hy` не используется.

9 Алгоритмы STL

В данной главе описаны все алгоритмы стандартной библиотеки STL. Глава начинается с общего обзора алгоритмов и принципов их работы. Затем приводятся точные сигнатуры всех алгоритмов и примеры их использования.

Заголовочные файлы алгоритмов

Чтобы использовать алгоритмы стандартной библиотеки C++, необходимо включить в программу заголовочный файл `<algorithm>`¹:

```
#include <algorithm>
```

В этом заголовочном файле также определяются вспомогательные функции `min()`, `max()` и `swap()`, представленные на с. 79 и 81. Итераторная функция `iter_swap()` описана на с. 269.

Некоторые алгоритмы STL, предназначенные для обработки числовых данных, определяются в заголовочном файле `<numeric>`²:

```
#include <numeric>
```

Вообще говоря, математическим компонентам стандартной библиотеки C++ посвящена глава 12, однако численные алгоритмы рассматриваются в этой главе. На взгляд автора, то, что они являются алгоритмами STL, важнее их числовой специализации.

При работе с алгоритмами также часто применяются объекты функций и функциональные адаптеры. Они были описаны в главе 8, а их определения находятся в файле `<functional>`³:

```
#include <functional>
```

¹ В исходной версии STL все алгоритмы определялись в заголовочном файле `<algo.h>`.

² В исходной версии STL все алгоритмы определялись в заголовочном файле `<algo.h>`.

³ В исходной версии STL объекты функций и функциональные адаптеры определялись в заголовочном файле `<function.h>`.

Общий обзор алгоритмов

В этом разделе приведен обзор всех алгоритмов стандартной библиотеки C++. Этот обзор даст читателю представление о возможностях алгоритмов и поможет выбрать лучший алгоритм для решения конкретной проблемы.

Введение

Алгоритмы уже были представлены в главе 5. На с. 106 и 121 рассматривалась роль алгоритмов и некоторые важные ограничения, связанные с их использованием. Любой алгоритм STL работает с одним или несколькими интервалами, заданными при помощи итераторов. Для первого интервала обычно задаются обе границы (начало и конец), а для остальных интервалов часто достаточно одного начала, потому что конец определяется количеством элементов в первом интервале. Перед вызовом необходимо убедиться в том, что заданные интервалы действительны, то есть начало интервала предшествует концу или совпадает с ним, а оба итератора относятся к одному контейнеру. Кроме того, дополнительные интервалы должны содержать достаточное количество элементов.

Алгоритмы работают в режиме замены, а не в режиме вставки, поэтому перед вызовом алгоритма необходимо убедиться в том, что приемный интервал содержит достаточное количество элементов. Специальные итераторы вставки (см. с. 275) переводят алгоритм в режим вставки.

Для повышения мощи и гибкости некоторые алгоритмы позволяют передавать пользовательские операции, которые вызываются при внутренней работе алгоритма. Такие операции оформляются в виде функций или объектов функций. Функция, возвращающая логическое значение, называется *предикатом*. Предикаты применяются в следующих ситуациях.

- Функция (или объект функции), определяющая унарный предикат, может передаваться алгоритму поиска в качестве критерия поиска. Унарный предикат проверяет, соответствует ли элемент заданному критерию. Например, это позволяет найти первый элемент со значением, меньшим 50.
- Функция (или объект функции), определяющая бинарный предикат, может передаваться алгоритму сортировки в качестве критерия сортировки. Бинарный предикат сравнивает два элемента. Например, с помощью бинарного предиката можно отсортировать объекты, представляющие людей, по фамилиям и по именам (см. пример на с. 296).
- Унарный предикат может использоваться как критерий, определяющий, к каким элементам должна применяться операция. Например, из коллекции можно удалить только элементы с нечетными значениями.
- Предикаты также используются для модификации операций в численных алгоритмах. Например, алгоритм `accumulate()`, обычно вычисляющий сумму элементов, также позволяет вычислять произведения всех элементов.

Помните, что состояние предиката не должно изменяться вследствие вызова функции (см. с. 303).

Примеры функций и объектов функций, передаваемых в виде параметров алгоритмов, приводятся на с. 129 и 134, а также в главе 8.

Классификация алгоритмов

Разные алгоритмы предназначены для решения разных задач, поэтому их можно классифицировать по основным областям применения. Например, одни алгоритмы только читают данные, другие модифицируют их, а третья изменяют порядок следования элементов. В этом подразделе кратко охарактеризованы возможности каждого алгоритма и их отличия от других похожих алгоритмов.

По названию алгоритма можно получить первое представление о его назначении. Проектировщики STL ввели два специальных суффикса.

- *Суффикс _if.* Суффикс `_if` используется при наличии двух похожих форм алгоритма с одинаковым количеством параметров; первой форме передается значение, а второй — функция или объект функции. В этом случае версия без суффикса `_if` используется при передаче значения, а версия с суффиксом `_if` — при передаче функции или объекта функции. Например, алгоритм `find()` ищет элемент с заданным значением, а алгоритм `find_if()` — элемент, удовлетворяющий критерию, определенному в виде функции или объекта функции.

Впрочем, не все алгоритмы, получающие функции и объекты функций, имеют суффикс `_if`. Если такая версия вызывается с дополнительными аргументами, отличающими ее от других версий, за ней сохраняется прежнее имя. Например, версия алгоритма `min_element()` с двумя аргументами находит в интервале минимальный элемент, при этом элементы сравниваются оператором `<`. В версии `min_element()` с тремя аргументами третий аргумент определяет критерий сравнения.

- *Суффикс _copy.* Суффикс `_copy` означает, что алгоритм не только обрабатывает элементы, но и копирует их в приемный интервал. Например, алгоритм `reverse()` переставляет элементы интервала в обратном порядке, а `reverse_copy()` копирует элементы в другой интервал в обратном порядке.

Приводимые ниже описания алгоритмов делятся на группы:

- немодифицирующие алгоритмы;
- модифицирующие алгоритмы;
- алгоритмы удаления;
- перестановочные алгоритмы;
- алгоритмы сортировки;
- алгоритмы упорядоченных интервалов;
- численные алгоритмы.

Если алгоритм принадлежит сразу нескольким категориям, он описывается в той категории, которую автор считает наиболее важной.

Немодифицирующие алгоритмы

Немодифицирующие алгоритмы сохраняют как порядок следования обрабатываемых элементов, так и их значения. Они работают с итераторами ввода и прямыми итераторами и поэтому могут вызываться для всех стандартных контейнеров. В табл. 9.1 перечислены алгоритмы стандартной библиотеки STL, не изменяющие состояния контейнера. На с. 330 приведен список немодифицирующих алгоритмов, предназначенных для упорядоченных входных интервалов.

Таблица 9.1. Немодифицирующие алгоритмы

Название	Описание	Страница
for_each()	Выполняет операцию с каждым элементом	333
count()	Возвращает количество элементов	337
count_if()	Возвращает количество элементов, удовлетворяющих заданному критерию	337
min_element()	Возвращает элемент с минимальным значением	338
max_element()	Возвращает элемент с максимальным значением	338
find()	Ищет первый элемент с заданным значением	340
find_if()	Ищет первый элемент, удовлетворяющий заданному критерию	340
search_n()	Ищет первые n последовательных элементов с заданными свойствами	342
search()	Ищет первое вхождение подинтервала	342
find_end()	Ищет последнее вхождение подинтервала	347
find_first_of()	Ищет первый из нескольких возможных элементов	349
adjacent_find()	Ищет два смежных элемента, равных по заданному критерию	351
equal()	Проверяет, равны ли два интервала	352
mismatch()	Возвращает первый различающийся элемент в двух интервалах	354
lexicographical_compare()	Проверяет, что один интервал меньше другого по лексикографическому критерию	356

Один из важнейших алгоритмов `for_each()` для каждого элемента вызывает операцию, переданную при вызове. Обычно операция выполняет некоторую обработку на уровне отдельных элементов. Например, при вызове `for_each()` можно передать функцию, которая выводит значение элемента. Кроме того, алгоритм `for_each()` позволяет вызывать для каждого элемента модифицирующую операцию. Следовательно, `for_each()` можно отнести как к модифицирующим, так и к немодифицирующим алгоритмам. Впрочем, старайтесь по возможности вместо `for_each()` использовать другие алгоритмы, реализованные специально для конкретных задач.

Некоторые немодифицирующие алгоритмы предназначены для поиска элементов. К сожалению, имена поисковых алгоритмов выглядят недостаточно логично, к тому же они отличаются от имен аналогичных строковых функций (табл. 9.2). Как это часто бывает, путаница возникала по историческим причинам. Во-первых, STL и строковые классы проектировались независимо друг от друга. Во-вторых, алгоритмы `find_end()`, `find_first_of()` и `search_n()` не входили в исходную версию STL. Например, имя `find_end()` вместо `search_end()` было выбрано случайно (занимаясь деталями, легко упустить некоторые аспекты общей картины, например логическую согласованность). Также случайно выяснилось, что форма `search_n()` нарушает общие концепции исходной версии STL. Проблема описана на с. 344.

Таблица 9.2. Сравнение строковых функций и алгоритмов поиска

Поиск	Строковая функция	Алгоритм STL
Первое вхождение одного элемента	<code>find()</code>	<code>find()</code>
Последнее вхождение одного элемента	<code>rfind()</code>	<code>find()</code> с обратным итератором
Первое вхождение подинтервала	<code>find()</code>	<code>search()</code>
Последнее вхождение подинтервала	<code>rfind()</code>	<code>find_end()</code>
Первое вхождение нескольких элементов	<code>find_first_of()</code>	<code>find_first_of()</code>
Последнее вхождение нескольких элементов	<code>find_last_of()</code>	<code>find_first_of()</code> с обратным итератором
Первое вхождение n последовательных элементов		<code>search_n()</code>

Модифицирующие алгоритмы

Модифицирующие алгоритмы изменяют значения элементов. Модификация производится непосредственно внутри интервала или в процессе копирования в другой интервал. Если элементы копируются в приемный интервал, исходный интервал остается без изменений. В табл. 9.3 перечислены модифицирующие алгоритмы стандартной библиотеки C++.

Таблица 9.3. Модифицирующие алгоритмы

Название	Описание	Страница
<code>for_each()</code>	Выполняет операцию с каждым элементом	333
<code>copy()</code>	Копирует интервал, начиная с первого элемента	358
<code>copy_backwards()</code>	Копирует интервал, начиная с последнего элемента	358
<code>transform()</code>	Модифицирует (и копирует) элементы; объединяет элементы двух интервалов	363
<code>merge()</code>	Производит слияние двух интервалов	406
<code>swap_ranges()</code>	Меняет местами элементы двух интервалов	365

Название	Описание	Страница
fill()	Заменяет каждый элемент заданным значением	366
fill_n()	Заменяет n элементов заданным значением	366
generate()	Заменяет каждый элемент результатом операции	368
generate_n()	Заменяет n элементов результатом операций	368
replace()	Заменяет элементы с заданным значением другим значением	369
replace_if()	Заменяет элементы, соответствующие критерию, заданным значением	369
replace_copy()	Заменяет элементы с заданным значением при копировании интервала	370
replace_copy_if()	Заменяет элементы, соответствующие критерию, при копировании интервала	370

В группе модифицирующих алгоритмов центральное место занимают алгоритмы `for_each()` (снова) и `transform()`. Оба алгоритма используются для модификации элементов последовательности, однако работают они по-разному.

- Алгоритму `for_each()` передается операция, которая модифицирует его аргумент. Это означает, что аргумент должен передаваться по ссылке. Пример:

```
void square (int& elem) // Передача по ссылке
{
    elem = elem * elem; // Прямое присваивание вычисленного значения
}
...
for_each(coll.begin(), coll.end(), // Интервал
        square); // Операция
```

- Алгоритм `transform()` использует операцию, которая возвращает модифицированный аргумент. Результат присваивается исходному элементу. Пример:

```
void square (int elem) // Передача по значению
{
    return elem * elem; // Возврат вычисленного значения
}
...
for_each(coll.begin(), coll.end(), // Источник
        coll.begin(), // Приемник
        square); // Операция
```

Алгоритм `transform()` работает чуть медленнее, потому что требует возвращения и присваивания результата вместо прямой модификации элемента. С другой стороны, этот подход более гибок, поскольку он также может использоваться для модификации элементов в процессе копирования в другой интервал. Кроме того, у алгоритма `transform()` есть еще одна версия, которая позволяет обрабатывать и комбинировать элементы из двух разных интервалов.

Строго говоря, алгоритм `merge()` мог бы и не входить в список модифицирующих алгоритмов. Он требует сортировки своих интервалов-источников, поэтому

его правильнее было бы включить в список алгоритмов упорядоченных интервалов (см. с. 330). Но на практике алгоритм `merge()` также успешно справляется со слиянием неупорядоченных интервалов. Конечно, результат тоже оказывается неупорядоченным. И все же для надежности стоит вызывать `merge()` только для упорядоченных интервалов.

Учтите, что элементы ассоциативных алгоритмов объявляются константными, чтобы алгоритм не мог нарушить порядок следования элементов вследствие модификации их значений. Это означает, что ассоциативные контейнеры не могут выступать в качестве приемника для модифицирующих алгоритмов.

Кроме модифицирующих алгоритмов в стандартной библиотеке C++ выделена отдельная группа модифицирующих алгоритмов для упорядоченных интервалов. Подробности приведены на с. 330.

Алгоритмы удаления

Алгоритмы удаления составляют отдельную подгруппу модифицирующих алгоритмов. Они предназначены для удаления элементов либо в отдельном интервале, либо в процессе копирования в другой интервал. Как и в случае с модифицирующими алгоритмами, их приемником не может быть ассоциативный контейнер, поскольку элементы ассоциативного контейнера считаются константными. В табл. 9.4 перечислены алгоритмы удаления стандартной библиотеки C++.

Таблица 9.4. Алгоритмы удаления

Название	Описание	Страница
<code>remove()</code>	Удаляет элементы с заданным значением	371
<code>remove_if()</code>	Удаляет элементы по заданному критерию	371
<code>remove_copy()</code>	Копирует элементы, значение которых отлично от заданного	373
<code>remove_copy_if()</code>	Копирует элементы, не соответствующие заданному критерию	373
<code>unique()</code>	Удаляет смежные дубликаты (элементы, равные своему предшественнику)	375
<code>unique_copy()</code>	Копирует элементы с удалением смежных дубликатов	377

Учтите, что алгоритмы удаления ограничиваются только «логическим» удалением элементов, то есть их перезаписью следующими элементами, которые не были удалены. Таким образом, количество элементов в интервалах, с которыми работают алгоритмы, остается прежним, а алгоритм возвращает позицию нового «логического конца» интервала. Вызывающая сторона должна использовать ее по своему усмотрению (например, физически уничтожить удаленные элементы). Подробная информация по этой теме приведена на с. 122.

Перестановочные алгоритмы

Перестановочными алгоритмами называются алгоритмы, изменяющие порядок следования элементов (но не их значения) посредством присваивания и пере-

становки их значений. В табл. 9.5 перечислены перестановочные алгоритмы стандартной библиотеки C++. Как и в случае с модифицирующими алгоритмами, их приемником не может быть ассоциативный контейнер, поскольку элементы ассоциативного контейнера считаются константными.

Таблица 9.5. Перестановочные алгоритмы

Название	Описание	Страница
reverse()	Переставляет элементы в обратном порядке	379
reverse_copy()	Копирует элементы, переставленные в обратном порядке	379
rotate()	Производит циклический сдвиг элементов	380
rotate_copy()	Копирует элементы с циклическим сдвигом	381
next_permutation()	Переставляет элементы	383
prev_permutation()	Переставляет элементы	383
random_shuffle()	Переставляет элементы в случайном порядке	385
partition()	Изменяет порядок следования элементов так, что элементы, соответствующие критерию, оказываются спереди	387
stable_partition()	То же, что и partition(), но с сохранением относительного расположения элементов, соответствующих и не соответствующих критерию	387

Алгоритмы сортировки

Алгоритмы сортировки являются частным случаем перестановочных алгоритмов, поскольку они тоже изменяют порядок следования элементов. Тем не менее сортировка является более сложной операцией и обычно занимает больше времени, чем простые перестановки. На практике эти алгоритмы обычно имеют сложность выше линейной¹ и требуют поддержки итераторов произвольного доступа (для приемника). Алгоритмы сортировки перечислены в табл. 9.6.

Таблица 9.6. Алгоритмы сортировки

Название	Описание	Страница
sort()	Сортирует все элементы	389
stable_sort()	Сортирует с сохранением порядка следования равных элементов	389
partial_sort()	Сортирует до тех пор, пока первые n элементов не будут упорядочены правильно	391
nth_element()	Сортирует элементы слева и справа от элемента в позиции n	394
partition()	Изменяет порядок следования элементов так, что элементы, соответствующие критерию, оказываются спереди	387

продолжение ↴

¹ Сложность операций рассматривается на с. 37.

Таблица 9.6 (продолжение)

Название	Описание	Страница
stable_partition()	То же, что и partition(), но с сохранением относительного расположения элементов, соответствующих и не соответствующих критерию	387
make_heap()	Преобразует интервал в кучу	397
push_heap()	Добавляет элемент в кучу	397
pop_heap()	Удаляет элемент из кучи	397
sort_heap()	Сортирует кучу (которая после вызова перестает быть кучей)	397

Алгоритмы сортировки часто критичны по времени, поэтому стандартная библиотека C++ содержит несколько алгоритмов, различающихся по способу сортировки или составу сортируемых элементов (полная/частичная сортировка). Например, алгоритм `nth_element()` прекращает работу, когда n -й элемент последовательности занимает правильное место в соответствии с критерием сортировки. Для остальных элементов он гарантирует только то, что предшествующие элементы имеют меньшее либо равное, а последующие элементы — большее либо равное значение. Сортировка всех элементов осуществляется перечисленными ниже алгоритмами.

- `sort()`. Исторически этот алгоритм основан на механизме быстрой сортировки, гарантирующем хорошую среднестатистическую сложность ($n \times \log(n)$), но очень плохую (квадратичную) сложность в худшем случае:

```
/* Сортировка всех элементов
 * - средняя сложность  $n \times \log(n)$ 
 * - квадратичная сложность  $n^2$  в худшем случае
 */
sort (coll.begin(), coll.end());
```

Если в вашей ситуации очень важно избежать худших случаев, воспользуйтесь другим алгоритмом, например `partial_sort()` или `stable_sort()`.

- `partial_sort()`. Исторически этот алгоритм основан на механизме сортировки в куче (heapsort), гарантирующем сложность $n \times \log(n)$ в любом случае. Тем не менее обычно сортировка в куче выполняется в 2–5 раз медленнее быстрой сортировки. Итак, с учетом реализаций `sort()` и `partial_sort()`, алгоритм `partial_sort()` лучше по сложности, но по скорости работы `sort()` обычно преосходит его. Преимущество алгоритма `partial_sort()` заключается в том, что сложность $n \times \log(n)$ гарантирована и никогда не ухудшается до квадратичной сложности.

Алгоритм `partial_sort()` также обладает особым свойством: он прекращает сортировку, когда требуется отсортировать только n первых элементов. Чтобы отсортировать все элементы, передайте конец последовательности во втором и в последнем аргументах:

```
/* Сортировка всех элементов
 * - всегда сложность  $n \times \log(n)$ 
```

```
* - но обычно работает вдвое медленнее sort()
*/
partial_sort (coll.begin(), coll.end(), coll.end());
```

- **stable_sort()**. Исторически этот алгоритм основан на механизме сортировки со слиянием. Он сортирует все элементы переданного интервала:

```
/* Сортировка всех элементов
 * - сложность  $n \log(n)$  или  $n \log(n) \log(n)$ 
 */
stable_sort (coll.begin(), coll.end());
```

Для достижения сложности $n \log(n)$ необходима дополнительная память, в противном случае алгоритм выполняется со сложностью $n \log(n) \times \log(n)$. Достоинством **stable_sort()** является сохранение порядка следования равных элементов.

Итак, теперь вы приблизительно представляете, какой алгоритм лучше всего подходит для ваших целей, однако это не все. Стандарт гарантирует лишь сложность, но не реализацию алгоритма. Это сделано для того, чтобы проектировщики могли использовать новые разработки в области алгоритмов и совершенствовать реализации без нарушения стандартов. Например, алгоритм **sort()** в реализации STL от SGI использует механизм интроспективной сортировки — новый алгоритм, который по умолчанию работает как быстрая сортировка, но в случаях с квадратичной сложностью переключается на сортировку в куче. Впрочем, у того факта, что стандарт не диктует реализацию алгоритма, есть и недостаток — можно реализовать очень плохой алгоритм, который соответствует стандарту. Например, реализация **sort()** на базе сортировки в куче будет считаться соответствующей стандарту. Конечно, оптимальный алгоритм можно выбрать на основании тестирования, однако нельзя гарантировать, что программный код хронометража будет работать на других платформах.

Помимо перечисленных существуют и другие алгоритмы сортировки элементов. Например, алгоритмы сортировки в куче вызывают функции, непосредственно работающие с кучей (то есть с бинарным деревом, используемым в реализации этих алгоритмов). Алгоритмы сортировки в куче заложены в основу эффективной реализации приоритетных очередей (см. с. 438). Вызовы этих алгоритмов, сортирующие все элементы коллекции, выглядят так:

```
/* сортировка всех элементов
 * - сложность  $n+n \log(n)$ 
 */
make_heap (coll.begin(), coll.end());
sort_heap (coll.begin(), coll.end());
```

За дополнительной информацией о кучах и алгоритмах сортировки в куче обращайтесь на с. 396.

Алгоритмы ***n*th_element()** существуют на тот случай, если вам нужен только *n*-й упорядоченный элемент или подмножество из *n* старших или младших эле-

ментов (неупорядоченное). Алгоритм `nth_element()` разделяет элементы на два подмножества в соответствии с критерием сортировки. Впрочем, задача также может быть решена алгоритмами `partition()` и `stable_partition()`. Различия между этими алгоритмами заключаются в следующем.

- Алгоритму `nth_element()` передается требуемое количество элементов в первой части (а следовательно, и во второй). Пример:

```
// Перемещение четырех наименьших элементов в начало
nth_element (coll.begin(), // Начало интервала
              coll.begin() + 3. // Позиция, отделяющая первую часть
                                // от второй
              coll.end()): // Конец интервала
```

Но после вызова невозможно сказать, по какому критерию первая часть отличается от второй. Более того, в обеих частях могут присутствовать элементы, совпадающие по значению с n -м элементом.

- Алгоритму `partition()` передается конкретный критерий сортировки, определяющий различия между первой и второй частями:

```
// Перемещение всех элементов, меньших 7, в начало
vector<int>::iterator pos;
pos = partition (coll.begin(), coll.end(),           // Интервал
                  bind2nd(less<int>(), 7));      // Критерий
```

На этот раз после вызова невозможно сказать, сколько элементов входит в первую и вторую части. Возвращаемый итератор `pos` указывает на первый элемент второй части, которая содержит все элементы, не соответствующие критерию.

- Алгоритм `stable_partition()` в целом аналогичен `partition()`, но он дополнительно гарантирует сохранение порядка следования элементов в обеих частях по отношению к другим элементам, входящим в ту же часть.

Любому алгоритму сортировки в необязательном аргументе всегда можно передавать критерий сортировки. По умолчанию критерием сортировки является объект функции `less<>`, а элементы сортируются по возрастанию значений.

Как и в случае с модифицирующими алгоритмами, приемником алгоритмов сортировки не может быть ассоциативный контейнер, поскольку элементы ассоциативного контейнера считаются константными.

Кроме того, алгоритмы сортировки не могут вызываться для списков, поскольку списки не поддерживают итераторы произвольного доступа. Впрочем, для сортировки элементов в списках определена функция `sort()` (см. с. 252).

Алгоритмы упорядоченных интервалов

Алгоритмы упорядоченных интервалов требуют, чтобы интервалы, с которыми они работают, были изначально упорядочены по соответствующему критерию. В табл. 9.7 перечислены все алгоритмы стандартной библиотеки C++, предназначенные для упорядоченных интервалов. Достоинством этих алгоритмов является невысокая сложность.

Таблица 9.7. Алгоритмы упорядоченных интервалов

Название	Описание	Страница
binary_search()	Проверяет, содержит ли интервал заданный элемент	400
includes()	Проверяет, что каждый элемент интервала также является элементом другого интервала	401
lower_bound()	Находит первый элемент со значением, большим либо равным заданному	402
upper_bound()	Находит первый элемент со значением, большим заданного	402
equal_range()	Возвращает количество элементов в интервале, равных заданному значению	404
merge()	Выполняет слияние элементов двух интервалов	406
set_union()	Вычисляет упорядоченное объединение двух интервалов	407
set_intersection()	Вычисляет упорядоченное пересечение двух интервалов	408
set_difference()	Вычисляет упорядоченный интервал, который содержит все элементы интервала, не входящие в другой интервал	409
set_symmetric_difference()	Вычисляет упорядоченный интервал, содержащий все элементы, входящие только в один из двух интервалов	410
inplace_merge()	Выполняет слияние двух последовательных упорядоченных интервалов	413

Первые пять алгоритмов упорядоченных интервалов, перечисленные в таблице, являются немодифицирующими, поскольку они ограничиваются поиском по заданному условию. Другие алгоритмы комбинируют два упорядоченных входных интервала и записывают результат в приемный интервал. Как правило, результат выполнения алгоритмов тоже упорядочен.

Численные алгоритмы

Численные алгоритмы выполняют разнообразную обработку числовых элементов. В табл. 9.8 приведен список численных алгоритмов стандартной библиотеки C++. Имена алгоритмов дают некоторое представление о том, что они делают, однако эти алгоритмы отличаются большей гибкостью и мощью, чем может показаться на первый взгляд. Например, алгоритм `accumulate()` по умолчанию вычисляет сумму элементов. Если элементами являются строки, то вычисляется их конкатенация. А если переключиться с оператора `+` на оператор `*`, алгоритм вычислит произведение элементов.

Алгоритмы `accumulate()` и `inner_product()` вычисляют и возвращают сводное значение без модификации интервалов. Другие алгоритмы записывают свои результаты в приемный интервал, количество элементов в котором соответствует количеству элементов в исходном интервале.

Таблица 9.8. Численные алгоритмы

Название	Описание	Страница
accumulate()	Объединяет все значения элементов (вычисляет сумму, произведение и т. д.)	414
inner_product()	Объединяет все элементы двух интервалов	416
adjacent_difference()	Объединяет каждый элемент с его предшественником	419
partial_sum()	Объединяет каждый элемент со всеми предшественниками	418

Вспомогательные функции

В оставшейся части этой главы приводятся подробные описания всех алгоритмов. Для каждого алгоритма дается минимум один пример. Следующие вспомогательные функции упрощают код примеров, чтобы читатель мог сосредоточиться на наиболее содержательных аспектах:

```
// algo/algostuff.hpp
#ifndef ALGOSTUFF_HPP
#define ALGOSTUFF_HPP

#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <set>
#include <map>
#include <string>
#include <algorithm>
#include <functional>
#include <numeric>

/* PRINT_ELEMENTS()
 * - вывод небязательной строки С, за которой выводятся
 * - все элементы коллекции coll, разделенные пробелами.
 */
template <class T>
inline void PRINT_ELEMENTS (const T& coll, const char* optcstr="")
{
    typename T::const_iterator pos;

    std::cout << optcstr;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}
```

```

/* INSERT_ELEMENTS (collection, first, last)
 * - заполнение коллекции значениями от first до last
 * - ВНИМАНИЕ: интервал НЕ ЯВЛЯЕТСЯ полуоткрытым
 */
template <class T>
inline void INSERT_ELEMENTS (T& coll, int first, int last)
{
    for (int i=first; i<=last; ++i) {
        coll.insert(coll.end(), i);
    }
}

#endif /*ALGOSTUFF_HPP*/

```

Сначала в файл `agostuff.hpp` включаются все заголовочные файлы, которые могут быть задействованы в примерах, чтобы их не пришлось включать отдельно. Затем определяются две вспомогательные функции.

- Функция `PRINT_ELEMENTS()` выводит все элементы контейнера, переданного в первом аргументе, разделяя их пробелами. Второй необязательный аргумент определяет префикс — строку, которая будет выводиться перед элементами (см. с. 128).
- Функция `INSERT_ELEMENTS()` вставляет элементы в контейнер, переданный в первом аргументе. Элементам присваиваются значения в интервале, границы которого определяются вторым и третьим аргументами. В данном случае включаются оба аргумента (поэтому интервал не является полуоткрытым).

Алгоритм `for_each`

Алгоритм `for_each()` обладает чрезвычайно гибкими возможностями. С его помощью можно производить разнообразную обработку и модификацию каждого элемента.

`UnaryProc`

`for_each (InputIterator beg, InputIterator end, UnaryProc op)`

- Вызывает унарный предикат `op(elem)` для каждого элемента `elem` в интервале `[beg,end]`.
- Возвращает копию `op` (измененную в процессе выполнения).
- Операция `op` может модифицировать элементы. На с. 325 алгоритм `for_each()` сравнивается с алгоритмом `transform()`, который решает ту же задачу, но несколько иным способом.
- Возвращаемые значения `op` игнорируются.
- Реализация алгоритма `for_each()` приведена на с. 135.
- Сложность линейная (`numberOfElements` вызовов `op`).

В следующем примере алгоритм `for_each()` вызывает для каждого элемента функцию `print()`, которая выводит текущее значение элемента.

```
// algo/foreach1.cpp
#include "algostuff.hpp"
using namespace std;

// Функция, вызываемая для каждого элемента
void print (int elem)
{
    cout << elem << ' ';
}

int main()
{
    vector<int> coll;
    INSERT_ELEMENTS(coll,1,9);

    // call print() for each element
    for_each (coll.begin(), coll.end(), // Интервал
              print); // Операция
    cout << endl;
}
```

Программа выводит следующий результат:

```
1 2 3 4 5 6 7 8 9
```

Чтобы вызвать для каждого элемента функцию его класса, необходимо использовать адаптеры `mem_fun`. За подробностями и примерами обращайтесь на с. 307.

В следующем примере объект функции модифицирует элементы в процессе перебора:

```
// algo/foreach2.cpp
#include "algostuff.hpp"
using namespace std;

// Объект функции прибавляет к значению элемента приращение.
// заданное при его инициализации
template <class T>
class AddValue {
private:
    T theValue; // Приращение
public:
    // Конструктор инициализирует приращение
    AddValue (const T& v) : theValue(v) {}

    // Суммирование выполняется "вызовом функции" для элемента
    void operator() (T& elem) const {
        elem += theValue;
    }
};
```

```

int main()
{
    vector<int> coll;
    INSERT_ELEMENTS(coll,1,9);

    // Прибавить к каждому элементу 10
    for_each (coll.begin(), coll.end(),
              AddValue<int>(10));           // Интервал
                                              // Операция
    PRINT_ELEMENTS(coll);

    // Прибавить к каждому элементу значение первого элемента
    for_each (coll.begin(), coll.end(),           // range
              AddValue<int>(*coll.begin())); // operation
    PRINT_ELEMENTS(coll);
}

```

Класс `AddValue()` определяет объекты функций, которые прибавляют к каждому элементу приращение, переданное при вызове конструктора. Используя объект функции, вы сможете определить величину приращения на стадии выполнения программы. Результат выполнения программы выглядит так:

```

11 12 13 14 15 16 17 18 19
22 23 24 25 26 27 28 29 30

```

Подробное описание этого примера приведено на с. 138. Задача также решается при помощи алгоритма `transform()` (см. с. 363):

```

transform (coll.begin(), coll.end(),           // Источник
          coll.begin(),                   // Приемник
          bind2nd(plus<int>(),10));     // Операция
...
transform (coll.begin(), coll.end(),           // Источник
          coll.begin(),                   // Приемник
          bind2nd(plus<int>(),*coll.begin())); // Операция

```

На с. 325 приведен общий сравнительный анализ алгоритмов `for_each()` и `transform()`.

В третьем примере используется значение, возвращаемое алгоритмом `for_each()`. Особое свойство `for_each()` — возвращение выполняемой операции — позволяет выполнить необходимые вычисления и вернуть результат в объекте функций.

```

// algo/foreach3.cpp
#include "algostuff.hpp"
using namespace std;

// Объект функции для вычисления среднего арифметического
class MeanValue {
private:
    long num;      // Счетчик элементов
    long sum;      // Сумма всех значений элементов

```

```

public:
    // Конструктор
    MeanValue () : num(0), sum(0) {
    }

    // "Вызов функции"
    // - обработка очередного элемента последовательности
    void operator() (int elem) {
        num++;           // Увеличение счетчика
        sum += elem;    // Прибавление значения
    }

    // Возвращение среднего арифметического
    // (с явным преобразованием типа)
    operator double() {
        return static_cast<double>(sum) / static_cast<double>(num);
    }
};

int main()
{
    vector<int> coll;
    INSERT_ELEMENTS(coll, 1, 8);

    // Вычисление и вывод среднего арифметического
    double mv = for_each (coll.begin(), coll.end(), // Интервал
                          MeanValue());           // Операция
    cout << "mean value: " << mv << endl;
}

```

Программа выводит следующий результат:

```
mean value: 4.5
```

Этот пример, хотя и в слегка измененной форме, подробно рассматривается на с. 301.

Немодифицирующие алгоритмы

Алгоритмы, описанные в этом разделе, работают с элементами, не изменяя их значений или порядка следования.

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 с заданным значением в интервале $[beg, end]$.
- Вторая форма подсчитывает в интервале $[beg, end)$ элементы, для которых унарный предикат $op(elem)$ возвращает `true`.
- Тип возвращаемого значения *difference_type* представляет тип разности итераторов:

```
typedef iterator_traits<InputIterator>::difference_type
```

Структура *iterator_traits* описана на с. 288.

- Предикат *op* не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Предикат *op* не должен изменять передаваемые аргументы.
- В классах ассоциативных контейнеров (множества, мультимножества, отображения и мультиотображения) определена похожая функция `count()` для подсчета элементов с заданным ключом (см. с. 239).
- Сложность линейная (*numberOfElements* сравнений или вызовов *op* соответственно).

Пример подсчета элементов по разным критериям:

```
// algo/count1.cpp
#include "algostuff.hpp"
using namespace std;

bool isEven (int elem)
{
    return elem % 2 == 0;
}

int main()
{
    vector<int> coll;
    int num;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // Подсчет элементов со значением 4
    num = count (coll.begin(), coll.end(),        // Интервал
                 4);                           // Значение
    cout << "number of elements equal to 4: " << num << endl;

    // Подсчет четных элементов
    num = count_if (coll.begin(), coll.end(),      // Интервал
                   isEven);                  // Критерий
    cout << "number of elements with even value: " << num << endl;

    // Подсчет элементов со значениями, большими 4
    num = count_if (coll.begin(), coll.end(),      // Интервал
                   bind2nd(greater<int>(),4)); // Критерий
```

```

    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 пользовательской функции `isEven()` можно воспользоваться выражением:

```
not1(bind2nd(modulus<int>(), 2))
```

За подробностями обращайтесь на с. 307.

Минимум и максимум

`InputIterator`

```
min_element (InputIterator beg, InputIterator end)
```

`InputIterator`

```
min_element (InputIterator beg, InputIterator end, CompFunc op)
```

`InputIterator`

```
max_element (InputIterator beg, InputIterator end)
```

`InputIterator`

```
max_element (InputIterator beg, InputIterator end, CompFunc op)
```

○ Все версии алгоритмов возвращают позицию минимального или максимального элемент в интервале $[beg, end]$.

○ Версии с двумя аргументами сравнивают элементы оператором `<`.

○ Операция *op* определяет критерий сравнения двух элементов:

op(elem1, elem2)

Если первый элемент меньше второго, операция должна возвращать `true`.

○ Если в заданном интервале существует несколько элементов с минимальными или максимальными значениями, алгоритмы возвращают первый найденный элемент.

○ Предикат *op* не должен изменять передаваемые аргументы.

○ Сложность линейная (*numberOfElements*-1 сравнений или вызовов *op* соответственно).

Следующая программа выводит минимальный и максимальный элементы `coll`, а также элементы с минимальным и максимальным абсолютным значением (модулем):

```
// 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,2,8);
    INSERT_ELEMENTS(coll,-3,5);

    PRINT_ELEMENTS(coll);

    // Вычисление и вывод минимума и максимума
    cout << "minimum: "
        << *min_element(coll.begin(),coll.end())
        << endl;
    cout << "maximum: "
        << *max_element(coll.begin(),coll.end())
        << endl;

    // Вычисление и вывод минимума и максимума
    // по абсолютным значениям
    cout << "minimum of absolute values: "
        << *min_element(coll.begin(),coll.end(),
                        absLess)
        << endl;
    cout << "maximum of absolute values: "
        << *max_element(coll.begin(),coll.end(),
                        absLess)
        << endl;
}
```

Результат выполнения программы:

```
2 3 4 5 6 7 8 -3 -2 -1 0 1 2 3 4 5
minimum: -3
maximum: 8
minimum of absolute values: 0
maximum of absolute values: 8
```

Обратите внимание: алгоритмы возвращают *позицию* минимального/максимального элемента, а не значение, поэтому при выводе используется унарный оператор *.

Поиск элементов

Поиск первого совпадающего элемента

```
InputIterator
find (InputIterator beg, InputIterator end, const T& value)
```

```
InputIterator
find_if (InputIterator beg, InputIterator end, UnaryPredicate op)
```

- Первая форма возвращает позицию первого элемента со значением *value* в интервале $[beg, end]$.
- Вторая форма возвращает позицию первого элемента в интервале $[beg, end]$, для которого предикат *op(elem)* возвращает true.
- Если подходящий элемент не найден, обе формы возвращают *end*.
- Предикат *op* не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Предикат *op* не должен изменять передаваемые аргументы.
- В упорядоченных интервалах рекомендуется использовать алгоритмы *lower_bound()*, *upper_bound()*, *equal_range()* и *binary_search()* (см. с. 399).
- В классах ассоциативных контейнеров (множества, мультимножества, отображения и мультиотображения) определена аналогичная функция *find()*, которая имеет логарифмическую, а не линейную сложность (см. с. 239).
- Сложность линейная (не более *number of Elements* сравнений или вызовов *op* соответственно).

В следующем примере алгоритм *find()* используется для поиска подинтервала, который начинается и заканчивается элементами со значением 4:

```
// 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: ");

    // Поиск первого элемента со значением 4
    list<int>::iterator pos1;
    pos1 = find (coll.begin(), coll.end(),      // Интервал
                4);                         // Значение

    /* Поиск второго элемента со значением 4
     * - поиск начинается после позиции первого найденного элемента */

```

```

 * - со значением 4 (если он есть)
 */
list<int>::iterator pos2;
if (pos1 != coll.end()) {
    pos2 = find (++pos1, coll.end(), // Интервал
                 4); // Значение
}

/* Вывод всех элементов от первого до второго элемента
 * со значением 4, включая оба элемента
 * - Итератор pos1 необходимо вернуть к позиции первого элемента
 * со значением 4 (если он есть)
 * - конец интервала определяется позицией за вторым элементом
 * со значением 4 (если он есть)
 */
if (pos1!=coll.end() && pos2!=coll.end()) {
    copy (--pos1, ++pos2,
          ostream_iterator<int>(cout, " "));
    cout << endl;
}
}

```

Чтобы найти второй элемент со значением 4, необходимо увеличить позицию первого найденного элемента `pos1`. Но следует помнить, что попытки выйти за конец коллекции приводят к непредсказуемым последствиям. Если вы не уверены в существовании элемента, перед увеличением проверьте значение, возвращаемое алгоритмом `find()`. 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()` может повторно вызываться для одного интервала с разными значениями. Тем не менее использование найденных позиций для определения интервалов требует осторожности, поскольку интервал может оказаться недействительным. Проблема действительности интервалов рассматривается на с. 109.

В следующем примере алгоритм `find_if()` используется для поиска элементов по совершенно иному критерию:

```

// algo/find2.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: ");

    cout << endl;
}

```

```

// Поиск первого элемента, большего 3
pos = find_if (coll.begin(), coll.end(),      // Интервал
                bind2nd(greater<int>(),3)); // Критерий

// Вывод позиции
cout << "the "
    << distance(coll.begin(),pos) + 1
    << ". element is the first greater than 3" << endl;

// Поиск первого элемента, кратного 3
pos = find_if (coll.begin(), coll.end(),
                not1(bind2nd(modulus<int>(),3)));

// Вывод позиции
cout << "the "
    << distance(coll.begin(),pos) + 1
    << ". element is the first divisible by 3" << endl;
}

```

При первом вызове `find()` простой объект функции в сочетании с адаптером `bind2nd` ищет первый элемент со значением, большим 3. При втором вызове передается более сложная комбинация для поиска первого элемента, делящегося на 3 без остатка.

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

```

На с. 131 приведен пример использования алгоритма `find()` для поиска первого простого числа.

Поиск первых n последовательных совпадений

`InputIterator
search_n (InputIterator beg, InputIterator end, Size count, const T& value)`

`InputIterator
search_n (InputIterator beg, InputIterator end,
 Size count, const T& value, BinaryPredicate op)`

- Первая форма возвращает позицию первого из `count` последовательных элементов в интервале `[beg,end)`, каждый из которых имеет значение `value`.
- Вторая форма возвращает позицию первого из `count` последовательных элементов в интервале `[beg,end)`, для которых бинарный предикат `op(elem,value)` возвращает `true`.
- Если подходящий элемент не найден, обе формы возвращают `end`.
- Предикат `op` не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Предикат `op` не должен изменять передаваемые аргументы.

- Эти алгоритмы не входили в исходную версию STL, а при их включении была допущена определенная небрежность. Использование бинарного предиката во второй форме (вместо унарного предиката) нарушает логическую согласованность исходной версии STL (см. комментарий на с. 344).
- Сложность линейная (не более $numberOfElements \times count$ сравнений или вызовов *op* соответственно).

Пример поиска трех последовательных элементов со значениями, большими либо равными 3:

```
// algo/searchn1.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll);

    // Поиск четырех последовательных элементов со значением 3
    deque<int>::iterator pos;
    pos = search_n (coll.begin(), coll.end(),      // Интервал
                    4,                          // Счетчик
                    3);                         // Значение

    // Вывод результата
    if (pos != coll.end()) {
        cout << "four consecutive elements with value 3 "
            << "start with " << distance(coll.begin(),pos) +1
            << ". element" << endl;
    }
    else {
        cout << "no four consecutive elements with value 3 found"
            << endl;
    }

    // Поиск четырех последовательных элементов со значением, большим 3
    pos = search_n (coll.begin(), coll.end(),      // Интервал
                    4,                          // Счетчик
                    3,                          // Значение
                    greater<int>());           // Критерий

    // Вывод результата
    if (pos != coll.end()) {
        cout << "four consecutive elements with value > 3 "
            << "start with " << distance(coll.begin(),pos) +1
            << ". element" << endl;
    }
    else {
```

```

        cout << "no four consecutive elements with value > 3 found"
        << endl;
    }
}

```

Результат выполнения программы:

```

1 2 3 4 5 6 7 8 9
no four consecutive elements with value 3 found
four consecutive elements with value > 3 start with 4. element

```

При использовании второй формы алгоритма `search_n()` возникает весьма неприятная проблема. Рассмотрим второй вызов `search_n()`:

```

pos = search_n (coll.begin(), coll.end(),      // Интервал
                4,                      // Счетчик
                3,                      // Значение
                greater<int>());       // Критерий

```

Такая семантика поиска элементов, удовлетворяющих заданному критерию, отличается от той, что используется в других компонентах STL. По канонам STL этот вызов должен выглядеть так:

```

pos = search_n_if (coll.begin(), coll.end(),      // Интервал
                    4,                      // Счетчик
                    bind2nd(greater<int>(),3)); // Критерий

```

К сожалению, никто не заметил этой нечлоследовательности при включении новых алгоритмов в стандарт (они не входили в исходную версию STL). Некоторые даже полагают, что версия с четырьмя аргументами более удобна, однако она требует бинарного предиката даже в том случае, когда по логике должно быть достаточно унарного предиката. Например, конструкция с пользовательской унарной предикатной функцией обычно выглядит так:

```

bool isPrime (int elem);
...
pos = search_n_if (coll.begin(), coll.end(),      // Интервал
                    4,                      // Счетчик
                    isPrime);               // Критерий

```

Однако на практике алгоритм требует, чтобы вы использовали бинарный предикат. Из-за этого приходится либо менять сигнатуру функции, либо писать функцию-оболочку:

```

bool binaryIsPrime (int elem1, int) {
    return isPrime(elem1);
}

pos = search_n (coll.begin(), coll.end(),      // Интервал
                4,                      // Счетчик
                0,                      // Фиктивное значение
                binaryIsPrime);        // Бинарный критерий

```

Поиск первого подинтервала

```
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 подинтервале интервала [*beg*,*end*), совпадающем с искомым интервалом [*searchBeg*,*searchEnd*]).
- В первой форме элементы найденного подинтервала должны быть равны элементам искомого интервала.
- Во второй форме вызов бинарного предиката *op*(*elem*,*searchElem*) для соответствующих элементов искомого интервала и подинтервала должен возвращать *true*.
- Если подходящий элемент не найден, обе формы возвращают *end*.
- Предикат *op* не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Предикат *op* не должен изменять передаваемые аргументы.
- Проблема поиска интервала по известным значениям начального и конечного элементов рассматривается на с. 109.
- Сложность линейная (не более *numberOfElements* × *numberOfSearchElements* сравнений или вызовов *op* соответственно).

Следующий пример показывает, как найти первое вхождение серии элементов внутри другой последовательности (сравните с примером использования алгоритма *find_end()* на с. 348):

```
// 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: ");
```

```

// Поиск первого вхождения subcoll в coll
deque<int>::iterator pos;
pos = search (coll.begin(), coll.end(),           // Интервал
              subcoll.begin(), subcoll.end()); // Подинтервал

// Цикл повторяется до тех пор, пока внутри coll
// успешно находится очередное вхождение subcoll
while (pos != coll.end()) {
    // Вывод позиции первого элемента
    cout << "subcoll found starting with element "
        << distance(coll.begin(), pos) + 1
        << endl;

    // Поиск следующего вхождения subcoll
    ++pos;
    pos = search (pos, coll.end(),           // Интервал
                  subcoll.begin(), subcoll.end()); // Подинтервал
}

```

Результат выполнения программы:

```

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

```

В следующем примере вторая форма алгоритма `search()` используется для поиска серии элементов, определяемых более сложным критерием. В данном случае ищется серия из четного, нечетного и вновь четного элементов:

```

// algo/search2.cpp
#include "algostuff.hpp"
using namespace std;

// Проверка элемента на четность
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 checkEven()
 * - проверка последовательности "чет нечет чет"
 */
bool checkEvenArgs[3] = { true, false, true };

// Поиск первого подинтервала в coll
vector<int>::iterator pos;
pos = search (coll.begin(), coll.end(),           // Интервал
              checkEvenArgs, checkEvenArgs+3, // Значения подинтервала
              checkEven);                  // Критерий

// Цикл повторяется до тех пор, пока внутри coll
// успешно находится очередное вхождение подинтервала
while (pos != coll.end()) {
    // Вывод позиции первого элемента
    cout << "subrange found starting with element "
        << distance(coll.begin(), pos) + 1
        << endl;

    // Поиск следующего подинтервала в coll
    pos = search (++pos, coll.end(),           // Интервал
                  checkEvenArgs, checkEvenArgs+3, // Значения
                  checkEven);                  // Критерий
}
}

```

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 подинтервала

```

ForwardIterator
find_end (ForwardIterator beg, ForwardIterator end,
          ForwardIterator searchBeg, ForwardIterator searchEnd)

```

```

ForwardIterator
find_end (ForwardIterator beg, ForwardIterator end,
          ForwardIterator searchBeg, ForwardIterator searchEnd,
          BinaryPredicate op)

```

- Обе формы возвращают позицию первого элемента в последнем подинтервале интервала $[beg, end]$, совпадающем с искомым интервалом $[searchBeg, searchEnd]$.
- В первой форме элементы найденного подинтервала должны быть равны элементам искомого интервала.

- Во второй форме вызов бинарного предиката $op(elem, searchElem)$ для соответствующих элементов искомого интервала и подинтервала должен возвращать `true`.
- Если подходящий элемент не найден, обе формы возвращают `end`.
- Предикат `op` не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Предикат `op` не должен изменять передаваемые аргументы.
- Проблема поиска интервала по известным значениям начального и конечного элементов рассматривается на с. 109.
- Эти алгоритмы не входили в исходную версию STL. К сожалению, им было присвоено имя `find_end()` вместо `search_end()`, что было бы гораздо логичнее, поскольку алгоритм поиска первого подинтервала называется `search()`.
- Сложность линейная (на более $numberOfElements \times numberofSearchElements$ сравнений или вызовов `op` соответственно).

Следующий пример показывает, как найти последнее вхождение серии элементов внутри другой последовательности (сравните с примером использования алгоритма `search()` на с. 345):

```
// 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: ");

    // Поиск последнего вхождения subcoll в coll
    deque<int>::iterator pos;
    pos = find_end (coll.begin(), coll.end(),           // Интервал
                    subcoll.begin(), subcoll.end()); // Подинтервал

    // Цикл повторяется до тех пор, пока внутри coll
    // успешно находится очередное вхождение подинтервала
    deque<int>::iterator end(coll.end());
    while (pos != end) {
        // Вывод позиции первого элемента
        cout << pos->value();
        pos = find_end (coll.begin(), coll.end(),
                        subcoll.begin(), subcoll.end());
    }
}
```

```

cout << "subcoll found starting with element "
      << distance(coll.begin(), pos) + 1
      << endl;

// Поиск следующего вхождения subcoll
end = pos;
pos = find_end (coll.begin(), end,           // Интервал
                  subcoll.begin(), subcoll.end()); // Подинтервал
}
}

```

Программа выводит следующий результат:

```

coll: 1 2 3 4 5 6 7 8 9
subcoll: 3 4 5 6
subcoll found starting with element 10
subcoll found starting with element 3

```

Что касается второй формы алгоритма, обратитесь к примеру использования второй формы алгоритма `search()` на с. 346 — алгоритм `find_end()` применяется аналогичным образом.

Поиск первого из нескольких возможных значений

`ForwardIterator`

```

find_first_of (ForwardIterator1 beg, ForwardIterator1 end,
                ForwardIterator2 searchBeg, ForwardIterator2 searchEnd)

```

`ForwardIterator`

```

find_first_of (ForwardIterator1 beg, ForwardIterator1 end,
                ForwardIterator2 searchBeg, ForwardIterator2 searchEnd,
                BinaryPredicate op)

```

- Первая форма возвращает позицию первого элемента в интервале $[beg, end]$, значение которого также встречается в интервале $[searchBeg, searchEnd]$.
- Вторая форма возвращает позицию первого элемента в интервале $[beg, end]$, для которого вызов предиката $op(elem, searchElem)$ с элементами интервала $[searchBeg, searchEnd]$ возвращает `true`.
- Если подходящий элемент не найден, обе формы возвращают *end*.
- Предикат *op* не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Предикат *op* не должен изменять передаваемые аргументы.
- Используя обратные итераторы, можно найти последнее из нескольких возможных значений.
- Алгоритмы не входили в исходную версию STL.
- Сложность линейная (не более $numberOfElements \times numberOfSearchElements$ сравнений или вызовов *op* соответственно).

Пример использования алгоритма `find_first_of()`:

```
// 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: ");

    // Поиск в coll первого вхождения элемента из searchcoll
    vector<int>::iterator pos;
    pos = find_first_of (coll.begin(), coll.end(),           // Интервал
                        searchcoll.begin(),          // Начало искомых значений
                        searchcoll.end());          // Конец искомых значений
    cout << "first element of searchcoll in coll is element "
        << distance(coll.begin(),pos) + 1
        << endl;

    // Поиск в coll последнего вхождения search элемента из searchcoll
    vector<int>::reverse_iterator rpos;
    rpos = find_first_of (coll.rbegin(), coll.rend(),           // Интервал
                          searchcoll.begin(),          // Начало искомых значений
                          searchcoll.end());          // Конец искомых значений
    cout << "last element of searchcoll in coll is element "
        << distance(coll.begin(),rpos.base())
        << endl;
}
```

Второй вызов использует обратные итераторы для поиска последнего элемента со значением, равным значению одного из элементов `searchcoll`. При выводе позиции элемента вызывается функция `base()`, которая преобразует обратный итератор в обычный (чтобы позиция отсчитывалась от начала коллекции). Обычно к результату `distance()` прибавляется 1, так как для первого элемента расстояние равно 0, однако `base()` смещает логическую позицию, на которую ссылается итератор, что приводит к желаемому эффекту (функция `base()` описана на с. 274).

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 равными значениями

```
InputIterator  
adjacent_find (InputIterator beg, InputIterator end)
```

```
InputIterator  
adjacent_find (InputIterator beg, InputIterator end,  
               BinaryPredicate op)
```

- Первая форма возвращает позицию первого элемента со значением следующего элемента.
- Вторая форма возвращает позицию первого элемента в интервале $[beg, end]$, для которого предикат $op(elem, nextElem)$ возвращает `true`.
- Если подходящий элемент не найден, обе формы возвращают `end`.
- Предикат `op` не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Предикат `op` не должен изменять передаваемые аргументы.
- Сложность линейная (не более $numberOfElements$ сравнений или вызовов `op` соответственно).

Следующая программа демонстрирует обе формы алгоритма `adjacent_find()`:

```
// algo/adjfind1.cpp  
#include "algostuff.hpp"  
using namespace std;  
  
// Функция проверяет, что значение следующего объекта вдвое больше  
// значения предыдущего  
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: ");  
  
    // Поиск первых двух элементов 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;
}

// Поиск первых двух элементов, для которых значение второго
// вдвое превышает значение первого
pos = adjacent_find (coll.begin(), coll.end(), // Интервал
                      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 `adjacent_find()` ищет элементы с одинаковыми значениями. Вторая форма при помощи функции `doubled()` проверяет, что значение второго элемента вдвое больше значения первого. 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

```

Сравнение интервалов

Проверка на равенство

```

bool
equal (InputIterator1 beg, InputIterator1 end,
       InputIterator2 cmpBeg)

bool
equal (InputIterator1 beg, InputIterator1 end,
       InputIterator2 cmpBeg,
       BinaryPredicate op)

```

- Первая форма проверяет полное совпадение элементов интервалов $[beg, end)$ и $[cmpBeg, \dots)$ и возвращает результат в виде логической величины.
- Вторая форма проверяет, что каждый вызов бинарного предиката $op(elem, cmpElem)$ для соответствующих элементов интервалов $[beg, end)$ и $[cmpBeg, \dots)$ возвращает `true`.
- Предикат *op* не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Предикат *op* не должен изменять передаваемые аргументы.
- Перед вызовом необходимо убедиться в том, что интервал, начинающийся с *cmpBeg*, содержит достаточное количество элементов.

- Для выявления различий в случае несовпадения интервалов следует использовать алгоритм `mismatch()` (см. с. 354).
- Сложность линейная (не более `numberOfElements` сравнений или вызовов `op` соответственно).

Следующий пример демонстрирует использование обеих форм алгоритма `equal()`. Первый вызов проверяет равенство элементов в двух интервалах, а второй вызов при помощи вспомогательной предикатной функции проверяет совпадение соответствующих элементов двух коллекций по четности/нечетности.

```
// 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: ");

    // Проверка равенства двух коллекций
    if (equal (coll1.begin(), coll1.end(), // Первый интервал
               coll2.begin())) {           // Второй интервал
        cout << "coll1 == coll2" << endl;
    }
    else {
        cout << "coll1 != coll2" << endl;
    }

    // Проверка соответствия четности/нечетности элементов
    if (equal (coll1.begin(), coll1.end(), // Первый интервал
               coll2.begin(),           // Второй интервал
               bothEvenOrOdd)) {       // Критерий сравнения
        cout << "even and odd elements correspond" << endl;
    }
    else {
        cout << "even and odd elements do not correspond" << endl;
    }
}
```

Программа 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
```

Поиск первого несовпадения

```
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 $[beg, end)$ и $[cmpBeg, \dots)$ первые два элемента интервалов, имеющие разные значения.
- Вторая форма возвращает $[beg, end)$ и $[cmpBeg, \dots)$ первые два элемента интервалов, для которых бинарный предикат $op(elem, cmpElem)$ возвращает `false`.
- Если различия в интервалах не найдены, возвращается пара из *end* и соответствующего элемента второго интервала. Однако следует учитывать, что это еще не означает равенства интервалов, потому что второй интервал может содержать больше элементов.
- Предикат *op* не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Предикат *op* не должен изменять передаваемые аргументы.
- Перед вызовом необходимо убедиться в том, что интервал, начинающийся с *cmpBeg*, содержит достаточное количество элементов.
- Для проверки равенства интервалов следует использовать алгоритм `equal()` (см. с. 352).
- Сложность линейная (не более *numberOfElements* сравнений или вызовов *op* соответственно).

Пример использования обеих форм алгоритма `mismatch()`:

```
// algo/mismal.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll1;
    list<int> coll2;

    INSERT_ELEMENTS(coll1, 1, 6);
```

```
for (int i=1; i<=16; i*=2) {
    coll2.push_back(i);
}
coll2.push_back(3);

PRINT_ELEMENTS(coll1, "coll1: ");
PRINT_ELEMENTS(coll2, "coll2: ");

// Поиск первого расхождения
pair<vector<int>::iterator, list<int>::iterator> values;
values = mismatch (coll1.begin(), coll1.end(), // Первый интервал
                  coll2.begin());           // Второй интервал
if (values.first == coll1.end()) {
    cout << "no mismatch" << endl;
}
else {
    cout << "first mismatch: "
        << *values.first << " and "
        << *values.second << endl;
}

/* Поиск первой позиции, в которой элемент coll1
 * не меньше соответствующего элемента coll2
 */
values = mismatch (coll1.begin(), coll1.end(), // Первый интервал
                  coll2.begin(),           // Второй интервал
                  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 `mismatch()` ищет первую позицию, в которой элементы двух интервалов имеют разные значения. Если такая позиция будет найдена, значения элементов направляются в стандартный выходной поток данных. Второй вызов ищет первую позицию, в которой элемент первой коллекции больше элемента второй коллекции, и возвращает их значения. Результат выполнения программы выглядит так:

```
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
```

Сравнение по лексикографическому критерию

```
bool
lexicographical_compare (InputIterator1 beg, InputIterator1 end,
                        InputIterator2 beg, InputIterator2 end)

bool
lexicographical_compare (InputIterator1 beg, InputIterator1 end,
                        InputIterator2 beg, InputIterator2 end,
                        CompFunc op)
```

- Обе формы проверяют, что интервал $[beg1, end1]$ меньше интервала $[beg2, end2]$ по лексикографическому критерию.
- Первая форма сравнивает элементы оператором $<$.
- Вторая форма сравнивает элементы бинарным предикатом $op(elem1, elem2)$. Предикат возвращает `true`, если $elem1$ меньше $elem2$.
- *Лексикографическое сравнение* означает, что интервалы сравниваются по парам элементов до тех пор, пока не будет выполнено одно из следующих условий:
 - если два элементы не равны, то результат их сравнения определяет результат всего сравнения;
 - интервал, содержащий меньшее количество элементов, считается меньше другого, то есть при достижении конца первого интервала результат сравнения считается равным `true` (первый интервал меньше второго);
 - если количество элементов в обоих интервалах одинаково, то интервалы считаются равными, а результат сравнения равен `false`.
- Предикат *op* не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Предикат *op* не должен изменять передаваемые аргументы.
- Сложность линейная (не более $2 \times \min(\text{numberOfElements1}, \text{numberOfElements2})$ сравнений или вызовов *op* соответственно).

Пример лексикографической сортировки коллекций:

```
// algo/lexicol.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.begin(), l1.end(), // Первый интервал
         l2.begin(), l2.end()); // Второй интервал
}
```

```
int main()
{
    list<int> c1, c2, c3, c4;

    // Заполнение коллекций одинаковыми исходными данными
    INSERT_ELEMENTS(c1, 1, 5);
    c4 = c3 = c2 = c1;

    // Внесение различий
    c1.push_back(7);
    c3.push_back(2);
    c3.push_back(0);
    c4.push_back(2);

    // Создание "коллекции коллекций"
    vector<list<int> > cc;

    cc.push_back(c1);
    cc.push_back(c2);
    cc.push_back(c3);
    cc.push_back(c4);
    cc.push_back(c3);
    cc.push_back(c1);
    cc.push_back(c4);
    cc.push_back(c2);

    // Вывод всех коллекций
    for_each (cc.begin(), cc.end(),
              printCollection);
    cout << endl;

    // Лексикографическая сортировка коллекции
    sort (cc.begin(), cc.end(),      // Интервал
          lessForCollection);      // Критерий сортировки

    // Повторный вывод коллекций
    for_each (cc.begin(), cc.end(),
              printCollection);
}
```

Вектор cc инициализируется несколькими списками. При вызове `sort()` коллекции сравниваются бинарным предикатом `lessForCollection()` (см. описание алгоритма `sort()` на с. 389). Алгоритм `lexicographical_compare()` производит лексикографическое сравнение коллекций. 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
```

Модифицирующие алгоритмы

В этом разделе описаны алгоритмы, модифицирующие элементы интервалов. Существуют два способа модификации:

- непосредственная модификация во время перебора;
- модификация в процессе копирования из источника в приемник.

Некоторые модифицирующие алгоритмы поддерживают оба способа модификации элементов в интервале. В этом случае форма с модификацией при копировании снабжается суффиксом `_copy`.

Ассоциативный контейнер не может быть приемником модифицирующего алгоритма, потому что элементы ассоциативных контейнеров являются константными. Без этого ограничения модификация привела бы к нарушению автоматической упорядоченности.

Все алгоритмы, использующие отдельный приемный интервал, возвращают позицию за последним скопированным элементом этого интервала.

Копирование элементов

```
OutputIterator
copy (InputIterator sourceBeg, InputIterator sourceEnd,
      OutputIterator destBeg)
```

```
BidirectionalIterator1
copy_backward (BidirectionalIterator1 sourceBeg,
              BidirectionalIterator1 sourceEnd,
              BidirectionalIterator2 destBeg)
```

- Оба алгоритма копируют все элементы исходного интервала $[sourceBeg, sourceEnd]$ в приемный интервал, начиная с $destBeg$ или заканчивая $destEnd$ соответственно.
- Алгоритмы возвращают позицию за последним скопированным элементом в приемном интервале (то есть позицию первого элемента, который не был заменен в ходе копирования).

- Итераторы *destBeg* и *destEnd* не могут входить в интервал $[sourceBeg, sourceEnd]$.
- Алгоритм *copy()* осуществляет перебор в прямом, а алгоритм *copy_backward()* — в обратном направлении. Данное различие существенно только в том случае, если источник и приемник перекрываются:
 - чтобы приемник перекрывал источник с начала, используйте алгоритм *copy()*, то есть для алгоритма *copy()* позиция *destBeg* должна предшествовать позиции *sourceBeg*;
 - чтобы приемник перекрывал источник с конца, используйте алгоритм *copy_backward()*, то есть для алгоритма *copy()* позиция *destBeg* должна находиться за позицией *sourceBeg*.

Итак, если третий аргумент ссылается на элемент исходного интервала, определяемого первыми двумя аргументами, используйте другой алгоритм. При этом вместо начала приемного интервала будет передаваться его конец. Пример на с. 374 демонстрирует различия между двумя алгоритмами.

- Алгоритма *copy_if()* не существует. Чтобы скопировать только те элементы, которые удовлетворяют заданному критерию, воспользуйтесь алгоритмом *remove_copy_if()* (см. с. 373).
- Чтобы скопировать элементы в обратном порядке, воспользуйтесь алгоритмом *reverse_copy()* (см. с. 379). Алгоритм *reverse_copy()* работает чуть эффективнее *copy()* с обратными итераторами.
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Примерная реализация алгоритма *copy()* приведена на с. 275.
- Копирование всех элементов контейнера для однотипных контейнеров осуществляется оператором присваивания (см. с. 241), для разнотипных контейнеров — функцией *assign()* (см. с. 241).
- Чтобы удалить элементы в процессе копирования, используйте алгоритмы *remove_copy()* и *remove_copy_if()* (см. с. 373).
- Чтобы модифицировать элементы в процессе копирования, используйте алгоритм *transform()* (см. с. 363) или *replace_copy()* (см. с. 370).
- Сложность линейная (*numberOfElements* присваиваний).

В следующем примере продемонстрированы простые вызовы *copy()*:

```
// algo/copy1.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll1;
    list<int> coll2;

    INSERT_ELEMENTS(coll1, 1, 9);
    copy(coll1.begin(), coll1.end(), coll2.begin());
```

```

/* Копирование элементов coll1 в coll2
 * - используем конечный итератор вставки,
 * чтобы вместо замены производилась вставка элементов
 */
copy (coll1.begin(), coll1.end(),           // Источник
      back_inserter(coll2));                // Приемник

/* Вывод элементов coll2
 * - копирование элементов в cout
 * с использованием потокового итератора вывода
 */
copy (coll2.begin(), coll2.end(),           // Источник
      ostream_iterator<int>(cout, " "));    // Приемник
cout << endl;

/* Копирование элементов coll1 в coll2 в обратном порядке
 * - на этот раз с заменой
 */
copy (coll1.rbegin(), coll1.rend(),         // Источник
      coll2.begin());                      // Приемник

// Повторный вывод элементов coll2
copy (coll2.begin(), coll2.end(),           // Источник
      ostream_iterator<int>(cout, " "));    // Приемник
cout << endl;
}

```

В этом примере конечный итератор вставки (см. с. 275) вставляет элементы в приемный интервал. Без применения итераторов вставки алгоритм `copy()` начинает замену в пустой коллекции `coll2`, что приводит к непредсказуемым последствиям. При использовании потоковых итераторов вывода (см. с. 281) в качестве приемника используется стандартный выходной поток.

Результат выполнения программы выглядит так:

```
1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1
```

Следующий пример демонстрирует различия между алгоритмами `copy()` и `copy_backward()`:

```

// algo/copy2.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
    /* Инициализация исходной коллекции символами
     * ".....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: ");

    // Копирование всех букв с позиции за три элемента перед 'a'
    vector<char> c1(source.begin(),source.end());
    copy (c1.begin()+10, c1.begin()+16, // Исходный интервал
          c1.begin()+7);           // Приемный интервал
    PRINT_ELEMENTS(c1,"c1:      ");

    // Копирование всех букв с позиции через три элемента после 'f'
    vector<char> c2(source.begin(),source.end());
    copy_backward (c2.begin()+10, c2.begin()+16, // Источник
                  c2.begin()+19);           // Приемник
    PRINT_ELEMENTS(c2,"c2:      ");
}

```

Обратите внимание: в обоих вызовах `copy()` и `copy_backward()` третий аргумент не входит в источник. Результат выполнения программы выглядит так:

```

source . . . . . . . . a b c d e f . . . . . .
c1:     . . . . . . . . a b c a b c d e f . . . . .
c2:     . . . . . . . . a b c a b c d e f . . . . .

```

В третьем примере алгоритм `copy()` обеспечивает фильтрацию данных между стандартными входным и выходным потоками. Программа читает строки и выводит их по одной в каждой строке:

```

// algo/copy3.cpp
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;

int main()
{
    copy (istream_iterator<string>(cin),           // Начало источника
          istream_iterator<string>(),               // Конец источника
          ostream_iterator<string>(cout, "\n"));   // Приемник
}

```

Преобразование и объединение элементов

Алгоритм `transform()` существует в двух вариантах:

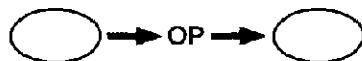
- первая форма вызывается с четырьмя аргументами и преобразует элементы при копировании из источника в приемник (то есть модификация осуществляется одновременно с копированием);
- вторая форма вызывается с пятью аргументами, она объединяет элементы из двух источников и записывает результат в приемник.

Преобразование элементов

`OutputIterator`

```
transform (InputIterator sourceBeg, InputIterator sourceEnd,
          OutputIterator destBeg,
          UnaryFunc op)
```

- Вызывает предикат `op(elem)` для каждого элемента в исходном интервале `[sourceBeg,sourceEnd)` и записывает каждый результат `op` в приемный интервал, начиная с `destBeg`:



- Возвращает позицию за последним преобразованным элементом в приемном интервале (то есть позицию первого элемента, не перезаписанного в результате операции).
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Позиции `sourceBeg` и `destBeg` могут быть идентичными. Это означает, что алгоритм `transform()`, как и `for_each()`, может использоваться для модификации элементов внутри интервала. Пример для алгоритма `for_each()` приведен на с. 325.
- Чтобы заменить заданной величиной элементы, удовлетворяющие критерию, используйте алгоритм `replace()` (см. с. 369).
- Сложность линейная (`numberOfElements` присваиваний).

Следующая программа демонстрирует использование данной формы алгоритма `transform()`:

```
// algo/transf1.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll1;
    list<int> coll2;

    INSERT_ELEMENTS(coll1,1,9);
    PRINT_ELEMENTS(coll1,"coll1:   ");

    // Изменение знака всех элементов coll1
    transform (coll1.begin(), coll1.end(),           // Источник
               coll1.begin(),                   // Приемник
               negate<int>());                // Операция
    PRINT_ELEMENTS(coll1,"negated: ");

    // Копирование элементов coll1, умноженных на 10, в coll2
    transform (coll1.begin(), coll1.end(),           // Источник
               coll2.begin(),                   // Приемник
               multiplies<int>(10));         // Операция
    PRINT_ELEMENTS(coll2,"coll2:   ");
}
```

```

        back_inserter(coll2),           // Приемник
        bind2nd(multiplies<int>(), 10)); // Операция
PRINT_ELEMENTS(coll2, "coll2:  ");

// Вывод элементов coll2 с изменением знака и в обратном порядке
transform (coll2.rbegin(), coll2.rend(),           // Источник
           ostream_iterator<int>(cout, " "), // Приемник
           negate<int>());                // Операция
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

```

На с. 315 приведен пример обработки элементов с применением двух разных операций.

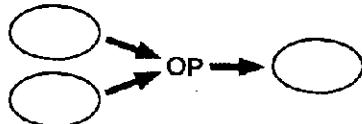
Комбинирование элементов двух интервалов

```

OutputIterator
transform (InputIterator1 sourceBeg, InputIterator1 sourceEnd,
          InputIterator2 source2Beg,
          OutputIterator destBeg,
          BinaryFunc op)

```

- Вызывает предикат $op(source1Elem, source2Elem)$ для каждой пары соответствующих элементов интервалов $[source1Beg, source1End]$ и $[source2Beg, \dots)$ и записывает каждый результат в приемный интервал, начиная с позиции $destBeg$.



- Возвращает позицию за последним преобразованным элементом в приемном интервале (то есть позицию первого элемента, не перезаписанного в результате операции).
- Перед вызовом необходимо убедиться в том, что второй источник имеет достаточный размер (то есть содержит не меньше элементов, чем первый источник).
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Позиции $source1Beg$, $source2Beg$ и $destBeg$ могут быть идентичными. Это означает, что вы можете вычислить комбинацию элементов самих с собой и записать результат на то же место.
- Сложность линейная ($numberOfElements$ вызовов $op()$).

Следующая программа демонстрирует использование данной формы алгоритма `transform()`:

```
// algo/transf2.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> col11;
    list<int> col12;

    INSERT_ELEMENTS(col11,1,9);
    PRINT_ELEMENTS(col11,"col11:    ");

    // Возведение элементов в квадрат
    transform (col11.begin(), col11.end(),           // Первый источник
               col11.begin(),                   // Второй источник
               col11.begin(),                   // Приемник
               multiplies<int>());            // Операция
    PRINT_ELEMENTS(col11,"squared: ");

    /* Сложить каждый элемент в порядке прямого перебора
     * с соответствующим элементом в порядке обратного перебора
     * и вставить результат в col12
     */
    transform (col11.begin(), col11.end(),           // Первый источник
               col11.rbegin(),                 // Второй источник
               back_inserter(col12),          // Приемник
               plus<int>());                // Операция
    PRINT_ELEMENTS(col12,"col12:    ");

    // Вывод разностей соответствующих элементов
    cout << "diff:    ";
    transform (col11.begin(), col11.end(),           // Первый источник
               col12.begin(),                 // Второй источник
               ostream_iterator<int>(cout, " "), // Приемник
               minus<int>());                // Операция
    cout << endl;
}
```

Результат выполнения программы выглядит так:

```
col11: 1 2 3 4 5 6 7 8 9
squared: 1 4 9 16 25 36 49 64 81
```

Обмен интервалов

```
ForwardIterator2
swap_ranges (ForwardIterator1 beg1, ForwardIterator1 end1,
             ForwardIterator2 beg2)
```

- Меняет местами элементы интервала $[beg1, end1]$ с соответствующими элементами интервала $[beg2, \dots]$.
- Возвращает позицию за последним переставленным элементом во втором интервале.
- Перед вызовом необходимо убедиться в том, что второй интервал имеет достаточный размер.
- Интервалы не должны перекрываться.
- Для обмена элементов в контейнерах одного типа рекомендуется использовать функцию `swap()` класса контейнера, поскольку она обычно выполняется с постоянной сложностью (с. 241).
- Сложность линейная (*numberOfElements* операций перестановки).

Пример использования алгоритма `swap_ranges()`:

```
// algo/swaps.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> col11;
    deque<int> col12;

    INSERT_ELEMENTS(col11, 1, 9);
    INSERT_ELEMENTS(col12, 11, 23);

    PRINT_ELEMENTS(col11, "col11: ");
    PRINT_ELEMENTS(col12, "col12: ");

    // Элементы col11 меняются местами с соответствующими элементами col12
    deque<int>::iterator pos;
    pos = swap_ranges (col11.begin(), col11.end(), // Первый интервал
                      col12.begin()); // Второй интервал

    PRINT_ELEMENTS(col11, "\ncol11: ");
    PRINT_ELEMENTS(col12, "col12: ");
    if (pos != col12.end()) {
        cout << "first element not modified: "
            << *pos << endl;
    }

    // Зеркальный обмен трех первых элементов col12
    // с тремя последними элементами
    swap_ranges (col12.begin(), col12.begin() + 3, // Первый интервал
                 col12.rbegin()); // Второй интервал

    PRINT_ELEMENTS(col12, "\ncol12: ");
}
```

Первый вызов `swap_ranges()` меняет местами элементы `coll1` с соответствующими элементами `coll2`. Остальные элементы `coll2` не изменяются. Алгоритм `swap_ranges()` возвращает позицию первого элемента, который остался без изменений. Второй вызов меняет местами первые и последние три элемента `coll2`. Один из итераторов является обратным, поэтому обмен происходит с зеркальным отображением элементов. 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
```

Присваивание

Заполнение интервала повторяющимся значением

```
void
fill (ForwardIterator beg, ForwardIterator end,
      const T& newValue)

void
fill_n (OutputIterator beg, Size num,
        const T& newValue)
```

- Алгоритм `fill()` присваивает значение `newValue` каждому элементу в интервале `[beg,end]`.
- Алгоритм `fill_n()` присваивает значение `newValue` первым `num` элементам в интервале, начинающемуся с `beg`.
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Сложность линейная (`numberOfElements` или `num` присваиваний соответственно).

Пример использования алгоритмов `fill()` и `fill_n()`.

```
// algo/fill1.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
    // Вывод десяти экземпляров 7.7
    fill_n(ostream_iterator<float>(cout, " "), // Начало приемника
           10,                                // Счетчик
           7.7);                             // Новое значение
```

```
cout << endl;

list<string> coll;

// Вставка девяти экземпляров "hello"
fill_n(back_inserter(coll),      // Начало приемника
       9,                      // Счетчик
       "hello");                // Новое значение
PRINT_ELEMENTS(coll, "coll: ");

// Замена всех элементов строкой "again"
fill(coll.begin(), coll.end(),   // Приемник
      "again");                // Новое значение
PRINT_ELEMENTS(coll, "coll: ");

// Замена всех элементов, кроме двух, строкой "hi"
fill_n(coll.begin(),           // Начало приемника
       coll.size()-2,          // Счетчик
       "hi");                  // Новое значение
PRINT_ELEMENTS(coll, "coll: ");

// Замена элементов от второго до предпоследнего строкой "hmmm"
list<string>::iterator pos1, pos2;
pos1 = coll.begin();
pos2 = coll.end();
fill (++pos1, --pos2,           // Приемник
      "hmmm");                // Новое значение
PRINT_ELEMENTS(coll, "coll: ");
}
```

При первом вызове алгоритм `fill_n()` используется для вывода определенного числа значений. Остальные вызовы `fill()` и `fill_n()` вставляют и заменяют значения в списке строк. Результат выполнения программы выглядит так:

```
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  
coll: again again again again again again again again  
coll: hi hi hi hi hi again again  
coll: hi hmmm hmmm hmmm hmmm hmmm hmmm again
```

Присваивание сгенерированных значений

```
void
generate (ForwardIterator beg, ForwardIterator end,
          Func op)

void
generate_n (OutputIterator beg, Size num,
            Func op)
```

- Алгоритм `generate()` присваивает значения, сгенерированные вызовом `op()` для каждого элемента в интервале $[beg, end]$.
- Алгоритм `generate_n()` присваивает значения, сгенерированные вызовом `op()` для первых *num* элементов в интервале $[beg, ...)$.
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Сложность линейная (*numberOfElements* или *num* вызовов `op()` и присваиваний).

В следующей программе алгоритмы `generate()` и `generate_n()` используются для вставки и присваивания случайных значений:

```
// algo/generate.cpp
#include <cstdlib>
#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;

    // Получение пяти случайных чисел
    generate_n (back_inserter(coll),      // Начало приемного интервала
                5,                      // Счетчик
                rand);                  // Генератор значений
    PRINT_ELEMENTS(coll);

    // Замена пятью новыми случайными числами
    generate (coll.begin(), coll.end(), // Приемный интервал
              rand);                  // Генератор значений
    PRINT_ELEMENTS(coll);
}
```

Функция `rand()` упоминается на с. 557. Примерный результат выполнения программы выглядит так:

```
41 18647 6334 26500 19169
15724 11478 29358 26962 24464
```

Результат зависит от платформы, так как последовательность случайных чисел, сгенерированная функцией `rand()`, не стандартизирована.

Пример использования алгоритма `generate()` с объектами функций для построения числовой последовательности приведен на с. 298.

Замена элементов

Замена внутри интервала

```
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 все элементы интервала $[beg, end]$, равные *oldValue*, значением *newValue*.
- Алгоритм `replace_if()` заменяет все элементы интервала $[beg, end]$, для которых унарный предикат *op(elem)* возвращает `true`, значением *newValue*.
- Предикат *op* не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Сложность линейная (*numberOfElements* сравнений или вызовов *op* соответственно).

Пример использования 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: ");

    // Замена всех элементов, равных 6, значением 42
    replace (coll.begin(), coll.end(),           // Интервал
              6,                           // Старое значение
              42);                         // Новое значение
    PRINT_ELEMENTS(coll, "coll: ");

    // Замена всех элементов, меньших 5, значением 0
    replace_if (coll.begin(), coll.end(),         // Интервал
                bind2nd(less<int>(), 5),        // Критерий замены
                0);                           // Новое значение
    PRINT_ELEMENTS(coll, "coll: ");
}
```

Результат выполнения программы выглядит так:

```
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
```

Замена при копировании

```
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 объединением алгоритмов `copy()` и `replace()`. Он заменяет каждый элемент интервала $[beg, end)$, равный *oldValue*, значением *newValue* в процессе копирования элементов в приемный интервал, начинаящийся с позиции *destBeg*.
- Алгоритм `replace_copy_if()` является объединением алгоритмов `copy()` и `replace_if()`. Он заменяет каждый элемент интервала $[beg, end)$, для которого унарный предикат *op(elem)* возвращает `true`, значением *newValue*. Замена производится в процессе копирования элементов в приемный интервал, начинаящийся с позиции *destBeg*.
- Оба алгоритма возвращают позицию за последним скопированным элементом в приемном интервале (то есть позицию первого элемента, который не был заменен в ходе копирования).
- Предикат *op* не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Сложность линейная (*numberOfElements* сравнений или вызовов *op* и присваиваний соответственно).

Пример использования алгоритмов `replace_copy()` и `replace_copy_if()`:

```
// algo/replace2.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll;
    INSERT_ELEMENTS(coll, 2, 6);
    INSERT_ELEMENTS(coll, 4, 9);
    PRINT_ELEMENTS(coll);

    // Вывод коллекции, в которой все элементы, равные 5,
    // заменены значением 55
    replace_copy(coll.begin(), coll.end(),           // Источник
                ostream_iterator<int>(cout, " "),      // Приемник
                5,                                         // Старое значение
                55);                                       // Новое значение
    cout << endl;

    // Вывод коллекции, в которой все элементы, меньше 5,
    // заменены значением 42
}
```

```
// print all elements with a value less than 5 replaced with 42
replace_copy_if(coll.begin(), coll.end(),           // Источник
                ostream_iterator<int>(cout, " "), // Приемник
                bind2nd(less<int>(), 5),        // Критерий замены
                42);                          // Новое значение
cout << endl;

// Вывод коллекции, в которой все нечетные элементы заменены нулями
replace_copy_if(coll.begin(), coll.end(),           // Источник
                ostream_iterator<int>(cout, " "), // Приемник
                bind2nd(modulus<int>(), 2),    // Критерий замены
                0);                           // Новое значение
cout << endl;
}
```

Результат выполнения программы выглядит так:

```
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
```

Алгоритмы удаления

Следующие алгоритмы удаляют элементы из интервала по заданному значению или критерию. Однако следует помнить, что эти алгоритмы *не могут* изменить количество элементов. Они всего лишь заменяют «удаленные» элементы следующими элементами, которые не были удалены, и возвращают новый логический конец интервала (позицию за последним элементом, который не был удален). За подробностями обращайтесь на с. 122.

Удаление элементов с заданным значением

Удаление элементов в интервале

ForwardIterator
remove (ForwardIterator *beg*, ForwardIterator *end*,
const T& *value*)

ForwardIterator
remove_if (ForwardIterator *beg*, ForwardIterator *end*,
UnaryPredicate *op*)

- Алгоритм *remove()* удаляет в интервале $[beg, end)$ все элементы, значение которых равно *value*.
- Алгоритм *remove_if()* удаляет в интервале $[beg, end)$ все элементы, для которых унарный предикат *op(elem)* возвращает *true*.

- Оба алгоритма возвращают новый логический конец модифицированного интервала (то есть позицию за последним элементом, который не был удален).
- Алгоритмы заменяют «удаленные» элементы следующими элементами, которые не были удалены.
- Алгоритмы сохраняют порядок следования элементов, которые не были удалены.
- Вызывающая сторона должна использовать новый логический конец интервала вместо исходного конца *end* (за дополнительной информацией обращайтесь на с. 122).
- Предикат *op* не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Алгоритм *remove_if()* обычно копирует унарный предикат внутри алгоритма и использует его дважды. Из-за этого могут возникнуть проблемы, если предикат изменяет свое состояние при вызове функции. За подробностями обращайтесь на с. 303.
- Вследствие модификации эти алгоритмы не могут использоваться с ассоциативными контейнерами (см. с. 125), однако в классах ассоциативных контейнеров определена аналогичная функция *erase()* (см. с. 248).
- Для списков определена аналогичная функция *remove()*, которая часто работает более эффективно, поскольку она меняет внутренние указатели вместо присваивания значений элементам (см. с. 247).
- Сложность линейная (*numberOfElements* сравнений или вызовов *op* соответственно).

Пример использования 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:      ");

    // Удаление всех элементов со значением 5
    vector<int>::iterator pos;
    pos = remove(coll.begin(), coll.end(), // Интервал
                 5);                      // Удаляемое значение

    PRINT_ELEMENTS(coll, "size not changed:  ");
```

```

// Стирание "удаленных" элементов из контейнера
coll.erase(pos, coll.end());
PRINT_ELEMENTS(coll, "size changed:      ");

// Удаление всех элементов со значением меньше 4
coll.erase(remove_if(coll.begin(), coll.end(), // Интервал
                     bind2nd(less<int>(), 4)), // Критерий удаления
            coll.end());
PRINT_ELEMENTS(coll, "<4 removed:      ");
}

```

Результат выполнения программы выглядит так:

```

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 объединением алгоритмов `copy()` и `remove()`. Он удаляет из исходного интервала $[beg, end)$ все элементы, равные `value`, в процессе копирования элементов в приемный интервал, начинающийся с позиции `destBeg`.
- Алгоритм `remove_copy_if()` является объединением алгоритмов `copy()` и `remove_if()`. Он удаляет каждый элемент интервала $[beg, end)$, для которого унарный предикат `op(elem)` возвращает `true`. Замена производится в процессе копирования элементов в приемный интервал, начинающийся с позиции `destBeg`.
- Оба алгоритма возвращают позицию за последним скопированным элементом в приемном интервале (то есть позицию первого элемента, который не был заменен).
- Предикат `op` не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Перед вызовом необходимо убедиться в том, что приемный интервал имсет достаточный размер, или использовать итераторы вставки.
- Сложность линейная (`numberOfElements` сравнений или вызовов `op` и присваиваний соответственно).

Пример использования алгоритмов `remove_copy()` и `remove_copy_if()`:

```
// algo/remove2.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
    list<int> coll1;

    INSERT_ELEMENTS(coll1,1,6);
    INSERT_ELEMENTS(coll1,1,9);
    PRINT_ELEMENTS(coll1);

    // Вывод содержимого коллекции кроме элементов со значением 3
    remove_copy(coll1.begin(), coll1.end(),           // Источник
                ostream_iterator<int>(cout, " "), // Приемник
                3);                                // Удаляемое значение
    cout << endl;

    // Вывод содержимого коллекции кроме элементов со значением, большим 4
    remove_copy_if(coll1.begin(), coll1.end(),          // Источник
                  ostream_iterator<int>(cout, " "), // Приемник
                  bind2nd(greater<int>(),4));     // Критерий удаления
    cout << endl;

    // Копирование всех элементов, больших 3, в мульти множество
    multiset<int> coll2;
    remove_copy_if(coll1.begin(), coll1.end(),          // Источник
                  inserter(coll2,coll2.end()), // Приемник
                  bind2nd(less<int>(),4));   // Критерий удаления
    PRINT_ELEMENTS(coll2);
}
```

Результат выполнения программы выглядит так:

```
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
```

Удаление дубликатов

Удаление смежных дубликатов

`ForwardIterator
unique (ForwardIterator beg, ForwardIterator end)`

`ForwardIterator
unique (ForwardIterator beg, ForwardIterator end,
BinaryPredicate op)`

- Обе формы «сворачивают» серии одинаковых элементов, оставляя начальный элемент и удаляя последующие дубликаты.
- Первая форма удаляет из интервала (beg, end) каждый элемент, равный предыдущему элементу. Таким образом, дубликаты удаляются только в том случае, если интервал был предварительно отсортирован (или, по крайней мере, если все элементы с одинаковыми значениями хранятся в смежных позициях).
- Вторая форма удаляет все элементы, следующие за элементом e , для которых бинарный предикат $\text{op}(\text{elem}, e)$ возвращает `true`. Иначе говоря, предикат используется для сравнения элемента не с предшественником, а с предыдущим элементом, который не был удален (см. пример).
- Обе формы возвращают новый логический конец модифицированного интервала (то есть позицию за последним элементом, который не был удален).
- Алгоритмы заменяют «удаленные» элементы следующими элементами, которые не были удалены.
- Алгоритмы сохраняют порядок следования элементов, которые не были удалены.
- Вызывающая сторона должна использовать новый логический конец интервала вместо исходного конца end (за дополнительной информацией обращайтесь на с. 122).
- Предикат op не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Вследствие модификации эти алгоритмы не могут использоваться с ассоциативными контейнерами (см. с. 125).
- Для списков определена аналогичная функция `unique()`, которая часто работает более эффективно, поскольку она меняет внутренние указатели вместо присваивания значений элементам (см. с. 250).
- Сложность линейная (numberOfElements сравнений или вызовов op соответственно).

Пример использования 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 };
    int sourceNum = sizeof(source)/sizeof(source[0]);

    list<int> coll;

    // Инициализация coll элементами source
    copy (source, source+sourceNum,           // Источник
          back_inserter(coll));             // Приемник
```

```

PRINT_ELEMENTS(coll);

// Удаление последовательных дубликатов
list<int>::iterator pos;
pos = unique (coll.begin(), coll.end());

/* Вывод оставшихся элементов
 * - с использованием нового логического конца
 */
copy (coll.begin(), pos,           // Источник
      ostream_iterator<int>(cout, " ")); // Приемник
cout << "\n\n";

// Повторная инициализация coll элементами source
copy (source, source+sourceNum,      // Источник
      coll.begin());                  // Приемник
PRINT_ELEMENTS(coll);

// Удаление элемента, если ему предшествует элемент с большим значением
coll.erase (unique (coll.begin(), coll.end(),
                    greater<int>()),
            coll.end());
PRINT_ELEMENTS(coll);
}

```

Результат выполнения программы выглядит так:

```

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 `unique()` уничтожает последовательные дубликаты. Второй вызов показывает, как работает вторая форма алгоритма. Из коллекции удаляются все элементы, следующие за текущим элементом, для которых сравнение критерием `greater` дает результат `true`. Например, первый элемент 6 больше следующих за ним элементов 1, 2, 2, 3 и 1, поэтому все эти элементы удаляются. Иначе говоря, предикат сравнивает элемент не с предшественником, а с предыдущим элементом, который не был удален (другой пример приведен далее при описании алгоритма `unique_copy()`).

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 объединением алгоритмов `copy()` и `unique()`.
- Обе формы копируют в приемный интервал, начинающийся с позиции `destBeg`, все элементы исходного интервала (beg, end) , за исключением смежных дубликатов.
- Обе формы возвращают позицию за последним скопированным элементом в приемном интервале (то есть позицию первого элемента, который не был заменен).
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Сложность линейная (`numberOfElements` сравнений или вызовов `op` и присваиваний соответственно).

Пример использования 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 };
    int sourceNum = sizeof(source)/sizeof(source[0]);

    // Инициализация coll элементами source
    list<int> coll;
    copy(source, source+sourceNum,           // Источник
          back_inserter(coll));           // Приемник
    PRINT_ELEMENTS(coll);

    // Вывод содержимого coll с удалением смежных дубликатов
    unique_copy(coll.begin(), coll.end(),           // Источник
                ostream_iterator<int>(cout, " ")); // Приемник
    cout << endl;

    // Вывод содержимого coll с удалением смежных элементов
    // со значениями, отличающимися на 1
    unique_copy(coll.begin(), coll.end(),           // Источник
                ostream_iterator<int>(cout, " "),      // Приемник
                differenceOne);                      // Критерий удаления
    cout << endl;
}
```

Результат выполнения программы выглядит так:

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
```

Обратите внимание на второй вызов `unique_copy()`. Он не удаляет элементы, отличающиеся от своего предшественника на 1, как могло бы показаться на первый взгляд. Вместо этого он удаляет все элементы, отличающиеся на 1 от предыдущего элемента, *который не был удален*. Например, после трех экземпляров значения 6 следующие элементы 5, 7 и 5 отличаются от 6 на 1, поэтому все они удаляются. Но следующие два экземпляра значения 4 остаются, поскольку при сравнении с 6 разность не равна 1.

В следующем примере сворачиваются последовательности пробелов:

```
// algo/unique3.cpp
#include <iostream>
#include <algorithm>
using namespace std;

bool bothSpaces (char elem1, char elem2)
{
    return elem1 == ' ' && elem2 == ' ';
}

int main()
{
    // По умолчанию начальные пробелы не пропускаются
    cin.unsetf(ios::skipws);

    /* Копирование стандартного ввода в стандартный вывод
     * - со сверткой пробелов
     */
    unique_copy(istream_iterator<char>(cin), // Начало источника: cin
                istream_iterator<char>(), // Конец источника: eof
                ostream_iterator<char>(cout), // Приемник: cout
                bothSpaces); // Критерий удаления
}
```

Входные данные:

```
Hello, here are sometimes more and sometimes fewer spaces.
```

Для этих входных данных программа выводит следующий результат:

```
Hello, here are sometimes more and sometimes fewer spaces.
```

Перестановочные алгоритмы

Перестановочные алгоритмы изменяют порядок следования элементов (но не их значения). Элементы ассоциативных контейнеров хранятся в фиксированном порядке, поэтому они не могут использоваться в качестве приемника для перестановочных алгоритмов.

Перестановка элементов в обратном порядке

```
void  
reverse (BidirectionalIterator beg, BidirectionalIterator end)  
  
OutputIterator  
reverse_copy (BidirectionalIterator sourceBeg,  
             BidirectionalIterator sourceEnd,  
             OutputIterator destBeg)
```

- Алгоритм `reverse()` переставляет элементы интервала $[beg, end)$ в обратном порядке.
- Алгоритм `reverse_copy()` переставляет в обратном порядке элементы интервала $[sourceBeg, sourceEnd)$ в процессе их копирования в приемный интервал, начинающийся с `destBeg`.
- Алгоритмы возвращают позицию за последним скопированным элементом в приемном интервале (то есть позицию первого элемента, который не был заменен в ходе копирования).
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Для списков определена аналогичная функция `reverse()`, которая часто работает более эффективно, поскольку она меняет внутренние указатели вместо присваивания значений элементам (см. с. 252).
- Сложность линейная ($numberOfElements/2$ обменов или $numberOfElements$ присваиваний соответственно).

Пример использования 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: ");  
  
    // Перестановка элементов в обратном порядке  
    reverse (coll.begin(), coll.end());  
    PRINT_ELEMENTS(coll, "coll: ");  
  
    // Перестановка в обратном порядке элементов  
    // со второго до предпоследнего  
    reverse (coll.begin() + 1, coll.end() - 1);  
    PRINT_ELEMENTS(coll, "coll: ");
```

```

    // Вывод всех элементов в обратном порядке
    reverse_copy (coll.begin(), coll.end(),           // Источник
                  ostream_iterator<int>(cout, " ")); // Приемник
    cout << endl;
}

```

Результат выполнения программы выглядит так:

```

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

```

Циклический сдвиг элементов

Циклический сдвиг элементов внутри интервала

- ```
void
rotate (ForwardIterator beg, ForwardIterator newBeg, ForwardIterator end)
```
- Производит циклический сдвиг элементов в интервале  $[beg, end]$ . После вызова  $*newBeg$  становится новым первым элементом.
  - Перед вызовом необходимо убедиться в том, что  $newBeg$  представляет действительную позицию в интервале  $[beg, end]$ ; в противном случае вызов приводит к непредсказуемым последствиям.
  - Сложность линейная (не более  $numberOfElements$  обменов).

Пример использования 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: ");

 // Циклический сдвиг на один элемент влево
 rotate (coll.begin(), // Начало интервала
 coll.begin() + 1, // Новый первый элемент
 coll.end()); // Конец интервала
 PRINT_ELEMENTS(coll, "one left: ");

 // Циклический сдвиг на два элемента вправо
 rotate (coll.begin(), // Начало интервала
 coll.end() - 2, // Новый первый элемент
 coll.end()); // Конец интервала
 PRINT_ELEMENTS(coll, "two right: ");
}

```

```

// Циклический сдвиг, в результате которого элемент со значением 4
// переходит в начало
rotate (coll.begin(), // Начало интервала
 find(coll.begin(), coll.end(), 4), // Новый первый элемент
 coll.end()); // Конец интервала
PRINT_ELEMENTS(coll, "4 first: ");
}

```

Как видно из приведенного примера, циклический сдвиг влево осуществляется с положительным смещением от начала, а сдвиг вправо — с отрицательным смещением от конца. Тем не менее прибавление смещения к итератору возможно только при использовании итераторов произвольного доступа (векторы поддерживают такие итераторы). Без итераторов произвольного доступа необходима функция `advance()` (см. далее пример применения функции `rotate_copy()`).

Результат выполнения программы выглядит так:

```

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)

```

- Алгоритм является объединением алгоритмов `copy()` и `rotate()`.
- Копирует элементы интервала  $[sourceBeg, sourceEnd)$  в приемный интервал, начинающийся с `destBeg`. При этом элементы циклически сдвигаются так, что `newBeg` становится новым первым элементом.
- Возвращает позицию за последним скопированным элементом в приемном интервале.
- Перед вызовом необходимо убедиться в том, что `newBeg` представляет элемент в интервале  $[beg, end)$ ; в противном случае вызов приводит к непредсказуемым последствиям.
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Источник не должен перекрываться с приемником.
- Сложность линейная (не более  $numberOfElements$  обменов).

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);

 // Вывод элементов с циклическим сдвигом на одну позицию влево
 set<int>::iterator pos = coll.begin();
 advance(pos, 1);
 rotate_copy(coll.begin(), // Начало источника
 pos, // Новый первый элемент
 coll.end(), // Конец источника
 ostream_iterator<int>(cout, " ")); // Приемник
 cout << endl;

 // Вывод элементов с циклическим сдвигом на две позиции вправо
 pos = coll.end();
 advance(pos, -2);
 rotate_copy(coll.begin(), // Начало источника
 pos, // Новый первый элемент
 coll.end(), // Конец источника
 ostream_iterator<int>(cout, " ")); // Приемник
 cout << endl;

 // Вывод элементов с циклическим сдвигом, в результате которого
 // элемент со значением 4 переходит в начало
 rotate_copy(coll.begin(), // Начало источника
 coll.find(4), // Новый первый элемент
 coll.end(), // Конец источника
 ostream_iterator<int>(cout, " ")); // Приемник
 cout << endl;
}

```

В отличие от приведенного ранее примера для алгоритма `rotate()` (см. с. 380) в данном случае вместо вектора используется множество. Смена контейнера влечет за собой два следствия:

- изменение итератора должно производиться функцией `advance()`, поскольку двунаправленные итераторы не поддерживают оператор `+`;
- вместо алгоритма `find()` должна использоваться более эффективная функция `find()`.

Результат выполнения программы выглядит так:

```

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

```

## Перестановка элементов

```
bool
next_permutation (BidirectionalIterator beg, BidirectionalIterator end)

bool
prev_permutation (BidirectionalIterator beg, BidirectionalIterator end)
```

- Алгоритм `next_permutation()` изменяет порядок следования элементов в интервале  $[beg, end)$  в соответствии со следующей перестановкой.
- Алгоритм `prev_permutation()` изменяет порядок следования элементов в интервале  $[beg, end)$  в соответствии с предыдущей перестановкой.
- Оба алгоритма возвращают `false`, если элементы находятся в «нормальном» (лексикографическом) порядке, то есть упорядочены по возрастанию для `next_permutation()` или по убыванию для `prev_permutation()`. Следовательно, для того чтобы перебрать все перестановки, нужно отсортировать элементы (по возрастанию или убыванию) и организовать цикл, который вызывает алгоритм `next_permutation()` или `prev_permutation()` до тех пор, пока алгоритм возвращает `true`<sup>1</sup>.

Лексикографическая сортировка рассматривается на с. 356.

- Сложность линейная (не более  $numberOfElements/2$  обменов).

Следующий пример показывает, как при помощи алгоритмов `next_permutation()` и `prev_permutation()` генерируются все перестановки элементов:

```
// algo/perm1.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
 vector<int> coll;
 INSERT_ELEMENTS(coll, 1, 3);
 PRINT_ELEMENTS(coll, "on entry: ");

 /* Генерировать перестановки элементов до тех пор, пока они
 * не будут упорядочены
 * - при этом перебираются все возможные перестановки,
 * - потому что в исходном состоянии
 * - элементы упорядочены по возрастанию
 */
 while (next_permutation(coll.begin(), coll.end())) {
 PRINT_ELEMENTS(coll, " ");
 }
}
```

---

<sup>1</sup> Алгоритмы `next_permutation()` и `prev_permutation()` также могут использоваться для сортировки элементов в интервале — просто вызывайте их до тех пор, пока возвращаемое значение остается равным `true`. Впрочем, такой способ сортировки отличается очень плохой эффективностью.

```

PRINT_ELEMENTS(coll,"afterward: ");

/* Генерировать перестановки элементов до тех пор,
 * пока элементы не будут упорядочены по убыванию
 * - это будет следующая перестановка после сортировки
 * - по возрастанию, поэтому цикл сразу же завершается
 */
while (prev_permutation(coll.begin(),coll.end())) {
 PRINT_ELEMENTS(coll," ");
}
PRINT_ELEMENTS(coll,"now: ");

/* Генерировать перестановки элементов до тех пор,
 * пока элементы не будут упорядочены по убыванию
 * - при этом перебираются все возможные перестановки,
 * - потому что в исходном состоянии
 * - элементы упорядочены по убыванию
 */
while (prev_permutation(coll.begin(),coll.end())) {
 PRINT_ELEMENTS(coll," ");
}
PRINT_ELEMENTS(coll,"afterward: ");
}

```

Результат выполнения программы выглядит так:

```

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

```

## Перестановка элементов в случайном порядке

```

void
random_shuffle (RandomAccessIterator beg, RandomAccessIterator end)

void
random_shuffle (RandomAccessIterator beg, RandomAccessIterator end,
 RandomFunc& op)

```

- Первая форма переставляет элементы интервала  $[beg, end)$  в случайном порядке, для чего используется генератор случайных чисел с равномерным распределением.
- Вторая форма переставляет элементы интервала  $[beg, end)$  с использованием операции  $op$ , вызываемой для целого значения типа `difference_type` итератора:  $op(max)$ . Операция  $op$  возвращает случайное число, большее либо равное 0, но меньшее  $max$  (возвращать  $max$  нельзя).
- Учтите, что операция  $op$  является неконстантной ссылкой, что исключает использование временных значений или обычных функций.
- Сложность линейная ( $numberOfElements - 1$  обменов).

Возможно, вас интересует, почему алгоритм `random_shuffle()` использует необязательную операцию как неконстантную ссылку. Дело в том, что генераторы случайных чисел обычно обладают локальным состоянием. Старые глобальные функции С (такие, как `rand()`) хранят свое локальное состояние в статических переменных. Однако такой подход имеет свои недостатки: в частности, генератор случайных чисел в принципе небезопасен по отношению к потокам выполнения (`threads`), поэтому вы не сможете сгенерировать два независимых потока данных (`streams`), состоящих из случайных чисел. По этой причине лучше воспользоваться объектами функций, локальное состояние которых инкапсулируется в переменных класса. Но это означает, что генераторы случайных чисел не могут быть константными, потому что они должны изменять свое локальное состояние при создании нового случайного числа. Чтобы генератор случайных чисел не был константным, его можно передать по значению вместо передачи по неконстантной ссылке. Но в этом случае генератор будет копироваться вместе со своим состоянием, и при каждой передаче генератора алгоритму будет генерироваться одна и та же псевдослучайная последовательность. С учетом всех перечисленных факторов было решено передавать генератор по неконстантной ссылке<sup>1</sup>.

Если вам потребуется использовать одну последовательность случайных чисел дважды, ее можно просто скопировать. Но если генератор основан на применении глобального состояния, вы будете получать разные последовательности.

Следующий пример демонстрирует случайную перестановку элементов с использованием алгоритма `random_shuffle()`:

```
// algo/random1.cpp
#include <cstdlib>
#include "algostuff.hpp"
using namespace std;

class MyRandom {
public:
 ptrdiff_t operator() (ptrdiff_t max) {
 double tmp;
```

<sup>1</sup> Спасибо Мэтту Остерну (Matt Austern) за это объяснение.

```

 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: ");

 // Случайная перестановка элементов
 random_shuffle (coll.begin(), coll.end());

 PRINT_ELEMENTS(coll,"shuffled: ");

 // Повторная сортировка
 sort (coll.begin(), coll.end());
 PRINT_ELEMENTS(coll,"sorted: ");

 /* Случайная перестановка элементов
 * с пользовательским генератором случайных чисел
 * - для передачи 1-значения необходимо использовать временный объект
 */
 MyRandom rd;
 random_shuffle (coll.begin(), coll.end(), // Интервал
 rd); // Генератор случайных чисел

 PRINT_ELEMENTS(coll,"shuffled: ");
}

```

При втором вызове алгоритма `random_shuffle()` используется пользовательский генератор случайных чисел `rd()`. Он представляет собой объект вспомогательного класса `MyRandom`, который задействует алгоритм построения случайных чисел, часто обеспечивающий лучший результат по сравнению с прямым вызовом `rand()`<sup>1</sup>.

Возможный (но не гарантированный) результат выполнения программы выглядит так:

```

coll: 1 2 3 4 5 6 7 8 9
shuffled: 2 6 9 5 4 3 1 7 8
sorted: 1 2 3 4 5 6 7 8 9
shuffled: 2 6 9 3 1 8 7 4 5

```

---

<sup>1</sup> Алгоритм построения случайных чисел, используемый в `MyRandom`, представлен и описан в книге Бъярна Страуструпа «*The C++ Programming Language*», 3rd Edition.

## Перемещение элементов в начало

```
BidirectionalIterator
partition (BidirectionalIterator beg, BidirectionalIterator end,
 UnaryPredicate op)
```

```
BidirectionalIterator
stable_partition (BidirectionalIterator beg, BidirectionalIterator end,
 UnaryPredicate op)
```

- Оба алгоритма перемещают в начало интервала  $[beg, end)$  все элементы, для которых унарный предикат  $op(elem)$  возвращает `true`.
- Оба алгоритма возвращают первую позицию, для которой  $op()$  возвращает `false`.
- Отличие `partition()` от `stable_partition()` заключается в том, что `stable_partition()` сохраняет относительный порядок следования элементов (как соответствующих, так и не соответствующих критерию).
- Алгоритм может использоваться для разбиения элементов на две группы в соответствии с критерием сортировки. Аналогичной возможностью обладает алгоритм `nth_element()`. Отличия этих алгоритмов от `nth_element()` описаны на с. 330.
- Предикат *op* не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Сложность:
  - `Partition()` — линейная (*numberOfElements* вызовов *op()* и не более *numberOfElements/2* обменов);
  - `Stable_partition()` — линейная при наличии дополнительной памяти (*numberOfElements* вызовов *op()* и обменов),  $n \times \log(n)$  в противном случае (*numberOfElements* $\times \log(\text{numberOfElements})$  вызовов *op()*).

Следующая программа демонстрирует использование алгоритмов `partition()` и `stable_partition()`, а также различия между ними:

```
// algo/part1.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
 vector<int> col11;
 vector<int> col12;

 INSERT_ELEMENTS(col11,1,9);
 INSERT_ELEMENTS(col12,1,9);
 PRINT_ELEMENTS(col11,"col11: ");
 PRINT_ELEMENTS(col12,"col12: ");
 cout << endl;

 // Перемещение всех нечетных элементов в начало
 vector<int>::iterator pos1, pos2;
```

```

pos1 = partition(coll1.begin(), coll1.end(), // Интервал
 not1(bind2nd(modulus<int>(), 2))); // Критерий
pos2 = stable_partition(coll2.begin(), coll2.end(), // Интервал
 not1(bind2nd(modulus<int>(), 2))); // Критерий

// Вывод коллекций и первого нечетного элемента
PRINT_ELEMENTS(coll1, "coll1: ");
cout << "first odd element: " << *pos1 << endl;
PRINT_ELEMENTS(coll2, "coll2: ");
cout << "first odd element: " << *pos2 << endl;
}

```

Результат выполнения программы выглядит так:

```

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 6 7 9
first odd element: 1

```

Как видно из приведенного примера, `stable_partition()`, в отличие от `partition()`, сохраняет относительный порядок следования четных и нечетных элементов.

## Алгоритмы сортировки

STL поддерживает несколько алгоритмов, предназначенных для сортировки элементов в интервалах. Кроме полной сортировки существует несколько разновидностей частичной сортировки. Если их результат вас устроит, лучше использовать эти алгоритмы, потому что обычно они работают более эффективно.

Ассоциативные контейнеры сортируют свои элементы автоматически. Однако обычно эффективнее однажды выполнить сортировку элементов, а не хранить их в упорядоченном виде постоянно (см. с. 232).

## 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 `sort()` и `stable_sort()` сортируют все элементы интервала  $[beg,end)$  оператором `<`.
- Вторые формы `sort()` и `stable_sort()` сортируют все элементы интервала, используя в качестве критерия сортировки бинарный предикат  $op(elem1, elem2)$ .
- Предикат  $op$  не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Отличие `sort()` от `stable_sort()` заключается в том, что `stable_sort()` сохраняет относительный порядок следования равных элементов.
- Алгоритмы не могут использоваться со списками, потому что списки не поддерживают итераторы произвольного доступа. Впрочем, для сортировки элементов в списках определена специальная функция `sort()` (см. с. 252).
- Алгоритм `sort()` гарантирует хорошее среднее быстродействие ( $n \times \log(n)$ ). Тем не менее если в вашей ситуации особенно важно избежать худших случаев, используйте алгоритм `partial_sort()` или `stable_sort()`. Сравнительный анализ алгоритмов сортировки приводится на с. 328.
- Сложность:
  - `Sort()` — в среднем  $n \times \log(n)$  (примерно  $numberOfElements \times \log(numberOfElements)$  сравнений);
  - `Stable_sort()` —  $n \times \log(n)$  при наличии дополнительной памяти ( $numberOfElements \times \log(numberOfElements)$ ),  $n \times \log(n) \times \log(n)$  в противном случае ( $numberOfElements \times \log(numberOfElements)^2$  сравнений).

Пример использования 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: ");

 // Сортировка элементов
 sort (coll.begin(), coll.end());

 PRINT_ELEMENTS(coll, "sorted: ");

 // Сортировка в обратном порядке
 sort (coll.begin(), coll.end(), // Интервал
 greater<int>()); // Критерий сортировки

 PRINT_ELEMENTS(coll, "sorted >: ");
}
```

Результат выполнения программы выглядит так:

```
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 сортировки, при которой в критерии используются функции класса элементов, приведен на с. 133.

Следующая программа демонстрирует различия между алгоритмами `sort()` и `stable_sort()`. В ней оба алгоритма сортируют строки по количеству символов, используя критерий `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()
{
 vector<string> col11;
 vector<string> col12;

 // Заполнение обеих коллекций одинаковыми элементами
 col11.push_back ("1xxx");
 col11.push_back ("2x");
 col11.push_back ("3x");
 col11.push_back ("4x");
 col11.push_back ("5xx");
 col11.push_back ("6xxxx");
 col11.push_back ("7xx");
 col11.push_back ("8xxx");
 col11.push_back ("9xx");
 col11.push_back ("10xxx");
 col11.push_back ("11");
 col11.push_back ("12");
 col11.push_back ("13");
 col11.push_back ("14xx");
 col11.push_back ("15");
 col11.push_back ("16");
 col11.push_back ("17");
 col12 = col11;

 PRINT_ELEMENTS(col11, "on entry:\n");

 // Сортировка по длине строк
 sort (col11.begin(), col11.end(), // Интервал
```

```

 lessLength); // Критерий
stable_sort (coll2.begin(), coll2.end(), // Интервал
 lessLength); // Критерий

PRINT_ELEMENTS(coll1, "\nwith sort():\n ");
PRINT_ELEMENTS(coll2, "\nwith stable_sort():\n ");
}

```

Результат выполнения программы выглядит так:

on entry:  
1xxx 2x 3x 4x 5xx 6xxxx 7xx 8xxx 9xx 10xxx 11 12 13 14xx 15 16 17

with sort():  
17 2x 3x 4x 16 15 13 12 11 9xx 7xx 5xx 8xxx 14xx 1xxx 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 следования элементов (начальные числа определяют порядок следования элементов в исходном состоянии коллекции).

## Частичная сортировка

### Простая 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 интервала  $[beg, end)$  оператором  $<$  так, чтобы интервал  $[beg, sortEnd)$  содержал упорядоченные элементы.
- Вторая форма сортирует элементы по критерию  $op(elem1, elem2)$  так, чтобы интервал  $[beg, sortEnd)$  содержал упорядоченные элементы.
- Предикат  $op$  не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- В отличие от `sort()` алгоритм `partial_sort()` сортирует не все элементы интервала, а прекращает сортировку после того, как подинтервал от начала до `sortEnd` будет заполнен упорядоченными элементами. Например, если после сортировки интервала вам понадобятся только первые три элемента, этот алгоритм работает более эффективно, поскольку он не тратит время на сортировку остальных элементов.

- Если *sortEnd* совпадает с *end*, алгоритм *partial\_sort()* сортирует весь интервал. В среднем по сложности он уступает *sort()*, но в худшем случае обеспечивает более высокое быстродействие (см. сравнительный анализ алгоритмов сортировки на с. 328).
- Сложность от линейной до  $n \times \log(n)$  (примерно  $\text{numberElements} \times \log(\text{numberElements})$  сравнений).

Пример использования алгоритма *partial\_sort()*:

```
// algo/pssort1.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);

 // Сортировка первых пяти элементов
 partial_sort (coll.begin(), // Начало интервала
 coll.begin() + 5, // Конец упорядоченного интервала
 coll.end()); // Конец всего интервала
 PRINT_ELEMENTS(coll);

 // Сортировка первых пяти элементов в обратном порядке
 partial_sort (coll.begin(), // Начало интервала
 coll.begin() + 5, // Конец упорядоченного интервала
 coll.end(), // Конец всего интервала
 greater<int>()); // Критерий сортировки
 PRINT_ELEMENTS(coll);

 // Сортировка всех элементов
 partial_sort (coll.begin(), // Начало интервала
 coll.end(), // Конец упорядоченного интервала
 coll.end()); // Конец всего интервала
 PRINT_ELEMENTS(coll);
}
```

Результат выполнения программы выглядит так:

```
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
```

## Частичная сортировка с копированием

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 являются объединениями алгоритмов `copy()` и `partial_sort()`.
- Обе формы копируют элементы из интервала  $[sourceBeg,sourceEnd)$  в приемный интервал  $[destBeg,destEnd)$ .
- Количество упорядоченных и скопированных элементов равно минимальному количеству элементов в исходном и приемном интервалах.
- Обе формы возвращают позицию за последним скопированным элементом в приемном интервале (то есть позицию первого элемента, который не был скопирован).
- Если количество элементов в приемном интервале  $[destBeg,destEnd)$  больше либо равно количеству элементов в источнике  $[sourceBeg,sourceEnd)$ , то алгоритм копирует и сортирует все элементы источника. В этом случае он работает как комбинация алгоритмов `copy()` и `sort()`.
- Сложность от линейной до  $n \times \log(n)$  (примерно  $numberOfElements \times \log(numberOfSortedElements)$  сравнений).

Пример использования алгоритма `partial_sort_copy()`:

```
// algo/psort2.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
 deque<int> col11;
 vector<int> col16(6); // Инициализация 6 элементами
 vector<int> col130(30); // Инициализация 30 элементами

 INSERT_ELEMENTS(col11,3,7);
 INSERT_ELEMENTS(col11,2,6);
 INSERT_ELEMENTS(col11,1,5);
 PRINT_ELEMENTS(col11);

 // Копирование упорядоченных элементов col11 в col16
 vector<int>::iterator pos6;
```

```

pos6 = partial_sort_copy (coll1.begin(), coll1.end(),
 coll6.begin(), coll6.end());

// Вывод всех скопированных элементов
copy (coll6.begin(), pos6,
 ostream_iterator<int>(cout, " "));
cout << endl;

// Копирование упорядоченных элементов coll1 в coll30
vector<int>::iterator pos30;
pos30 = partial_sort_copy (coll1.begin(), coll1.end(),
 coll30.begin(), coll30.end(),
 greater<int>());

// Вывод всех скопированных элементов
copy (coll30.begin(), pos30,
 ostream_iterator<int>(cout, " "));
cout << endl;
}

```

Результат выполнения программы выглядит так:

```

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

```

При первом вызове `partial_sort_copy()` приемник содержит только шесть элементов, поэтому алгоритм ограничивается копированием шести элементов и возвращает конец `coll6`. При втором вызове `partial_sort_copy()` все элементы `coll1` копируются в `coll30`; места для них достаточно, поэтому копирование и сортировка выполняются со всеми элементами.

## Разбиение 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 интервала  $[beg, end)$  так, чтобы элемент в позиции  $nth$  занимал правильное место в порядке сортировки, все предшествующие элементы были меньше либо равны этому элементу, а все последующие элементы — больше или равны ему. Таким образом, алгоритм создает два подинтервала, разделенных элементом в позиции  $nth$ , при этом любой элемент первого подинтервала меньше любого элемента второго подинтервала либо равен ему. Этого может быть достаточно, если вы хотите найти  $n$  старших или младших элементов без полной сортировки всего интервала.

- Первая форма сортирует элементы оператором `<`.
- Вторая форма сортирует элементы по критерию `op(elem1, elem2)`.
- Предикат `op` не должен изменять свое состояние во время вызова. За подробностями обращайтесь на с. 303.
- Алгоритм `partition()` (см. с. 387) тоже разбивает элементы интервала на две части по заданному критерию сортировки. Отличия между `partition()` и `nth_element()` описаны на с. 330.
- Сложность в среднем линейная.

Пример использования алгоритма `nth_element`:

```
// algo/nth1.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);

 // Выделение четырех наименьших элементов
 nth_element (coll.begin(), // Начало интервала
 coll.begin() + 3, // Границы элемент
 coll.end()); // Конец интервала

 // Вывод
 cout << "the four lowest elements are: ";
 copy (coll.begin(), coll.begin() + 4,
 ostream_iterator<int>(cout, " "));
 cout << endl;

 // Выделение четырех наибольших элементов
 nth_element (coll.begin(), // Начало интервала
 coll.end() - 4, // Границы элемент
 coll.end()); // Конец интервала

 // Вывод
 cout << "the four highest elements are: ";
 copy (coll.end() - 4, coll.end(),
 ostream_iterator<int>(cout, " "));
 cout << endl;

 // Выделение четырех наибольших элементов (второй вариант)
 nth_element (coll.begin(), // Начало интервала
```

```

 coll.begin() + 3, // Границный элемент
 coll.end(), // Конец интервала
 greater<int>()); // Критерий сортировки

 // Вывод
 cout << "the four highest elements are: ";
 copy (coll.begin(), coll.begin() + 4,
 ostream_iterator<int>(cout, " "));
 cout << endl;
}

```

Результат выполнения программы выглядит так:

```

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

```

## Сортировка в куче

В контексте сортировки *кучей* (*heap*) называется бинарное дерево, реализованное в виде последовательной коллекции. Кучи обладают двумя важными свойствами:

- первый элемент всегда является максимальным;
- добавление и удаление элементов производится с логарифмической сложностью.

Куча является идеальной основой для реализации приоритетных очередей (очередей, которые автоматически сортируют свои элементы), поэтому алгоритмы, работающие с кучей, используются контейнером *priority\_queue* (см. с. 438). В STL определены четыре алгоритма для работы с кучами:

- алгоритм *make\_heap()* преобразует интервал элементов в кучу;
- алгоритм *push\_heap()* добавляет новый элемент в кучу;
- алгоритм *pop\_heap()* удаляет элемент из кучи;
- алгоритм *sort\_heap()* преобразует кучу в упорядоченную коллекцию (которая после этого перестает быть кучей).

Как обычно, критерий сортировки может задаваться бинарным предикатом. По умолчанию сортировка осуществляется оператором *<*.

## Алгоритмы для работы с кучей

```

void
make_heap (RandomAccessIterator beg, RandomAccessIterator end)

void
make_heap (RandomAccessIterator beg, RandomAccessIterator end,
 BinaryPredicate op)

```

- Обе формы преобразуют элементы интервала  $[beg, end)$  в кучу.
- В необязательном параметре  $op$  передается бинарный предикат, определяющий критерий сортировки:  $op(elem1, elem2)$ .
- Алгоритмы необходимы лишь для начала работы с кучей, содержащей более одного элемента (один элемент автоматически является кучей).
- Сложность: линейная (не более  $3 \times \text{numberOfElements}$  сравнений).

```
void
push_heap (RandomAccessIterator beg, RandomAccessIterator end)
```

```
void
push_heap (RandomAccessIterator beg, RandomAccessIterator end,
 BinaryPredicate op)
```

- Обе формы добавляют последний элемент, находящийся перед  $end$ , в существующую кучу  $[beg, end-1]$ , в результате чего весь интервал  $[beg, end)$  становится кучей.
- В необязательном параметре  $op$  передается бинарный предикат, определяющий критерий сортировки:  $op(elem1, elem2)$ .
- Перед вызовом необходимо проследить за тем, чтобы элементы в интервале  $[beg, end-1)$  формировали кучу (с общим критерием сортировки), а новый элемент следовал непосредственно за ними.
- Сложность логарифмическая (не более  $\log(\text{numberOfElements})$  сравнений).

```
void
pop_heap (RandomAccessIterator beg, RandomAccessIterator end)
```

```
void
pop_heap (RandomAccessIterator beg, RandomAccessIterator end,
 BinaryPredicate op)
```

- Обе формы перемещают максимальный (то есть первый) элемент кучи  $[beg, end)$  в конец и создают новую кучу из оставшихся элементов в интервале  $[beg, end-1)$ .
- В необязательном параметре  $op$  передается бинарный предикат, определяющий критерий сортировки:  $op(elem1, elem2)$ .
- Перед вызовом необходимо проследить за тем, чтобы элементы в интервале  $[beg, end-1)$  формировали кучу (с общим критерием сортировки).
- Сложность логарифмическая (не более  $2 \times \log(\text{numberOfElements})$  сравнений).

```
void
sort_heap (RandomAccessIterator beg, RandomAccessIterator end)
```

```
void
sort_heap (RandomAccessIterator beg, RandomAccessIterator end,
 BinaryPredicate op)
```

- Обе формы преобразуют кучу  $[beg, end)$  в упорядоченный интервал.
- В необязательном параметре  $op$  передается бинарный предикат, определяющий критерий сортировки:  $op(elem1, elem2)$ .

- После вызова интервал перестает быть кучей.
- Перед вызовом необходимо проследить за тем, чтобы элементы в интервале  $[beg, end-1]$  формировали кучу (с общим критерием сортировки).
- Сложность логарифмическая (не более  $numberOfElements \times \log(numberOfElements)$  сравнений).

## Пример использования алгоритмов для работы с кучей

Следующая программа демонстрирует использование различных алгоритмов работы с кучей:

```
// 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: ");

 // Преобразование коллекции в кучу
 make_heap (coll.begin(), coll.end());

 PRINT_ELEMENTS (coll, "after make_heap(): ");

 // Извлечение следующего элемента из кучи
 pop_heap (coll.begin(), coll.end());
 coll.pop_back();

 PRINT_ELEMENTS (coll, "after pop_heap(): ");

 // Занесение нового элемента в кучу
 coll.push_back (17);
 push_heap (coll.begin(), coll.end());

 PRINT_ELEMENTS (coll, "after push_heap(): ");

 /* Преобразование кучи в упорядоченную коллекцию
 * - ВНИМАНИЕ: после вызова интервал перестает быть кучей
 */
 sort_heap (coll.begin(), coll.end());

 PRINT_ELEMENTS (coll, "after sort_heap(): ");
}
```

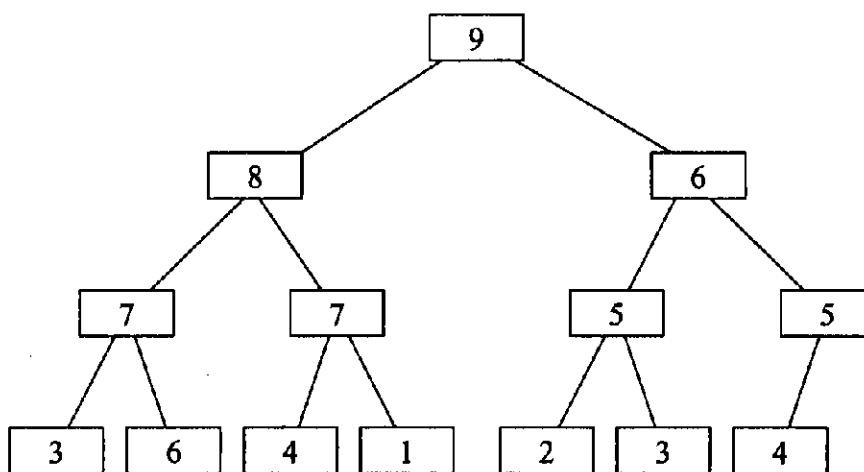
Результат выполнения программы выглядит так:

```
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 вызова `make_heap()` элементы сортируются в виде кучи:

9 8 6 7 7 5 5 3 6 4 1 2 3 4

Если преобразовать эту последовательность в бинарное дерево, вы увидите, что значение каждого узла меньше либо равно значению его родительского узла (рис. 9.1). Алгоритмы `push_heap()` и `pop_heap()` изменяют элементы так, что инвариант структуры бинарного дерева (любой узел не больше своего родительского узла) остается неизменным.



**Рис. 9.1.** Элементы кучи в виде бинарного дерева

## Алгоритмы упорядоченных интервалов

Алгоритмы упорядоченных интервалов требуют, чтобы элементы исходных интервалов были упорядочены по своим критериям сортировки. Обычно эти алгоритмы работают гораздо эффективнее своих аналогов для неупорядоченных интервалов (как правило, они обладают логарифмической, а не линейной сложностью). Алгоритмы упорядоченных интервалов могут использоваться с итераторами, которые не являются итераторами произвольного доступа, но в этом случае сложность оказывается линейной, поскольку им приходится последовательно перебирать элемент за элементом. Впрочем, количество сравнений все равно может оставаться логарифмическим.

Согласно стандарту, вызов этих алгоритмов для неупорядоченных интервалов приводит к непредсказуемым последствиям. Хотя в большинстве реализаций алгоритмы успешно работают с неупорядоченными последовательностями, использовать их в программе нельзя — это нарушит ее переносимость.

В ассоциативных контейнерах определены специальные функции, аналогичные по назначению представленным далее алгоритмам. При поиске по заданному значению или ключу следует использовать эти функции вместо алгоритмов.

## Поиск элементов

Следующие алгоритмы предназначены для поиска значений в упорядоченных интервалах.

### Проверка присутствия 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 интервал  $[beg, end)$  элемент со значением *value*.
- В необязательном параметре *op* передается бинарный предикат, определяющий критерий сортировки: *op(elem1, elem2)*.
- Чтобы определить позицию искомого элемента, воспользуйтесь функциями *lower\_bound()*, *upper\_bound()* и *equal\_range()* (см. с. 402).
- Перед вызовом интервал должен быть упорядочен по соответствующему критерию сортировки.
- Сложность логарифмическая для итераторов произвольного доступа, линейная в остальных случаях (не более  $\log(numberOfElements) + 2$  сравнений, но для итераторов, не являющихся итераторами произвольного доступа, выполняется линейное количество операций перебора, вследствие чего сложность оказывается линейной).

Пример использования алгоритма *binary\_search()*:

```
// algo/bsearch1.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
 list<int> coll;

 INSERT_ELEMENTS(coll, 1, 9);
 PRINT_ELEMENTS(coll);

 // Проверка существования элемента со значением 5
 if (binary_search(coll.begin(), coll.end(), 5)) {
 cout << "5 is present" << endl;
 }
}
```

```

else {
 cout << "5 is not present" << endl;
}

// Проверка существования элемента со значением 42
if (binary_search(coll.begin(), coll.end(), 42)) {
 cout << "42 is present" << endl;
}
else {
 cout << "42 is not present" << endl;
}
}

```

Результат выполнения программы выглядит так:

```

1 2 3 4 5 6 7 8 9
5 is present
42 is not present

```

## Проверка присутствия 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 интервал  $[beg, end]$  все элементы упорядоченного интервала  $[searchBeg, searchEnd]$ , и возвращают результат в виде логической величины. Иначе говоря, для каждого элемента интервала  $[searchBeg, searchEnd]$  должен существовать равный элемент в интервале  $[beg, end]$ . Если некоторые элементы интервала  $[searchBeg, searchEnd]$  равны, то в интервале  $[beg, end]$  должно присутствовать такое же количество дубликатов. Таким образом, интервал  $[searchBeg, searchEnd]$  должен быть подмножеством интервала  $[beg, end]$ .
- В необязательном параметре *op* передается бинарный предикат, определяющий критерий сортировки: *op(elem1, elem2)*.
- Перед вызовом оба интервала должны быть упорядочены по одному критерию сортировки.
- Сложность линейная (не более  $2 \times (numberOfElements + searchElements) - 1$  сравнений).

Пример использования алгоритма **Includes()**:

```

// algo/includes.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: ");

 // Проверка вхождения всех элементов search в coll
 if (includes (coll.begin(), coll.end(),
 search.begin(), search.end())) {
 cout << "all elements of search are also in coll"
 << endl;
 }
 else {
 cout << "not all elements of search are also in coll"
 << endl;
 }
}

```

Результат выполнения программы выглядит так:

```

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 со значением, большим либо равным *value*. Результат определяет первую позицию, в которой элемент со значением *value* вставляется без нарушения упорядоченности интервала  $[beg, end]$ .

- Алгоритм `upper_bound()` возвращает позицию первого элемента со значением, большим *value*. Результат определяет первую позицию, в которой элемент со значением *value* вставляется без нарушения упорядоченности интервала `[beg,end)`.
- Если позицию найти не удалось, оба алгоритма возвращают *end*.
- В необязательном параметре *op* передается бинарный предикат, определяющий критерий сортировки: `op(elem1,elem2)`.
- Перед вызовом интервал должен быть упорядочен по соответствующему критерию сортировки.
- Чтобы одновременно получить результаты алгоритмов `lower_bound()` и `upper_bound()`, воспользуйтесь алгоритмом `equal_range()`, который возвращает оба значения (см. далее).
- В ассоциативных контейнерах (множество, мультимножество, отображение и мультиотображение) определены эквивалентные функции, которые работают более эффективно (см. с. 239).
- Сложность логарифмическая для итераторов произвольного доступа, линейная в остальных случаях (не более  $\log(\text{numberOfElements})+1$  сравнений, но для итераторов, не являющихся итераторами произвольного доступа, выполняется линейное количество операций перебора, вследствие чего сложность оказывается линейной).

Пример использования алгоритмов `lower_bound()` и `upper_bound()`<sup>1</sup>:

```
// 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);

 // Вывод первой и последней позиций, в которых может быть
 // вставлен элемент со значением 5
 list<int>::iterator pos1, pos2;
 pos1 = lower_bound (coll.begin(), coll.end(),
 5);
 pos2 = upper_bound (coll.begin(), coll.end(),
 5);
```

<sup>1</sup> В прежних версиях STL может потребоваться файл `distance.hpp` (см. с. 268).

```

cout << "5 could get position "
 << distance(coll.begin(), pos1) + 1
 << " up to "
 << distance(coll.begin(), pos2) + 1
 << " without breaking the sorting" << endl;

// Вставка 3 в первой возможной позиции без нарушения упорядоченности
coll.insert(lower_bound(coll.begin(), coll.end(),
 3),
 3);

// Вставка 7 в первой возможной позиции без нарушения упорядоченности
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 4 4 5 5 6 6 7 7 7 8 8 9 9

```

## Поиск первой и последней возможных позиций

```

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 интервал значений, равных *value*. Интервал определяет первую и последнюю позиции, в пределах которых элемент со значением *value* вставляется без нарушения упорядоченности интервала *[beg,end)*.
- Алгоритм эквивалентен вызову
 

```
make_pair(lower_bound(...), upper_bound(...))
```
- В необязательном параметре *op* передается бинарный предикат, определяющий критерий сортировки: *op(elem1, elem2)*.
- Перед вызовом интервал должен быть отсортирован по соответствующему критерию сортировки.
- В ассоциативных контейнерах (множество, мультимножество, отображение и мультиотображение) определена эквивалентная функция, которая работает более эффективно (см. с. 240).
- Сложность логарифмическая для итераторов произвольного доступа, линейная в остальных случаях (не более  $2 \times \log(\text{number of Elements}) + 1$  сравнений,

но для итераторов, не являющихся итераторами произвольного доступа, выполняется линейное количество операций перебора, вследствие чего сложность оказывается линейной).

Пример использования алгоритмов `lower_bound()` и `upper_bound()`<sup>1</sup>:

```
// algo/eqrang.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);

 // Вывод первой и последней позиций, в которых может быть
 // вставлен элемент со значением 5
 pair<list<int>::iterator,list<int>::iterator> range;

 range = equal_range (coll.begin(), coll.end(),
 5);

 cout << "5 could get position "
 << distance(coll.begin(),range.first) + 1
 << " up to "
 << distance(coll.begin(),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 positions 9 up to 11 without breaking the sorting
```

## Слияние интервалов

Алгоритмы, представленные в этом разделе, комбинируют элементы двух интервалов и вычисляют результирующий итератор — сумму, объединение, пересечение и т. д.

### Суммирование двух упорядоченных множеств элементов

```
OutputIterator
merge (InputIterator source1Beg, InputIterator source1End,
 InputIterator source2Beg, InputIterator source2End,
 OutputIterator destBeg)
```

---

<sup>1</sup> В прежних версиях STL может потребоваться файл `distance.hpp` (см. с. 268).

`OutputIterator`

```
merge (InputIterator source1Beg, InputIterator source1End,
 InputIterator source2Beg, InputIterator source2End,
 OutputIterator destBeg, BinaryPredicate op)
```

- Обе формы комбинируют элементы упорядоченных исходных интервалов  $[source1Beg, source1End]$  и  $[source2Beg, source2End]$  таким образом, что приемный интервал  $[destBeg, ...)$  содержит все элементы как первого, так и второго интервалов. Например, рассмотрим два интервала:

```
1 2 2 4 6 7 7 9
2 2 2 3 6 6 8 9
```

В результате вызова алгоритма `merge()` для этих интервалов будет получен следующий интервал:

```
1 2 2 2 2 3 4 6 6 6 7 7 8 9 9
```

- В приемном интервале элементы следуют в порядке сортировки.
- Обе формы возвращают позицию за последним скопированным элементом в приемном интервале (то есть позицию первого элемента, который не был переписан).
- В необязательном параметре `op` передается бинарный предикат, определяющий критерий сортировки: `op(elem1, elem2)`.
- Алгоритмы не изменяют состояние исходных интервалов.
- Согласно стандарту, оба исходных интервала должны быть упорядочены перед вызовом. Впрочем, в большинстве реализаций алгоритм `merge()` также комбинирует неупорядоченные исходные интервалы в неупорядоченный приемный интервал. Но чтобы сохранить переносимость программы, вызов `merge()` следует заменить двукратным вызовом `copy()`.
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Приемный интервал не должен перекрываться с исходными интервалами.
- Для списков определена аналогичная функция `merge()`, комбинирующая элементы двух списков (см. с. 252).
- Чтобы элементы, присутствующие в обоих исходных интервалах, включались в приемный интервал только один раз, используйте алгоритм `set_union()` (см. с. 407).
- Чтобы в приемный интервал включались только элементы, присутствующие в обоих исходных интервалах, используйте алгоритм `set_intersection()` (см. с. 408).
- Сложность линейная (не более  $numberOfElements1 + numberOfElements2 - 1$  сравнений).

Пример использования алгоритма `merge()`:

```
// algo/merge1.cpp
#include "algostuff.hpp"
using namespace std;
```

```

int main()
{
 list<int> coll1;
 set<int> coll2;

 // Заполнение обеих коллекций упорядоченными элементами
 INSERT_ELEMENTS(coll1,1,6);
 INSERT_ELEMENTS(coll2,3,8);

 PRINT_ELEMENTS(coll1,"coll1: ");
 PRINT_ELEMENTS(coll2,"coll2: ");

 // Вывод результата слияния
 cout << "merged: ";
 merge (coll1.begin(), coll1.end(),
 coll2.begin(), coll2.end(),
 ostream_iterator<int>(cout, " "));
 cout << endl;
}

```

Результат выполнения программы выглядит так:

```

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

```

## Объединение двух упорядоченных множеств элементов

```

OutputIterator
set_union (InputIterator source1Beg, InputIterator source1End,
 InputIterator source2Beg, InputIterator source2End,
 OutputIterator destBeg)

set_union (InputIterator source1Beg, InputIterator source1End,
 InputIterator source2Beg, InputIterator source2End,
 OutputIterator destBeg, BinaryPredicate op)

```

- О Обе формы комбинируют элементы упорядоченных исходных интервалов  $[source1Beg,source1End]$  и  $[source2Beg,source2End]$  таким образом, что приемный интервал  $[destBeg,...)$  содержит все элементы, присутствующие в первом интервале, во втором интервале или в обоих интервалах сразу. Например, рассмотрим два интервала:

```

1 2 2 4 6 7 7 9
2 2 2 3 6 6 8 9

```

В результате вызова алгоритма `set_union()` для этих интервалов будет получен следующий интервал:

```

1 2 2 2 3 4 6 6 7 7 8 9

```

- О В приемном интервале элементы следуют в порядке сортировки.

- Элементы, входящие в оба интервала, включаются в объединение только в одном экземпляре. Тем не менее приемный интервал может содержать дубликаты, если соответствующий элемент дублируется в одном из исходных интервалов. Количество элементов с одинаковыми значениями в приемном интервале равно их максимальному количеству в двух исходных интервалах.
- Обе формы возвращают позицию за последним скопированным элементом в приемном интервале (то есть позицию первого элемента, который не был перезаписан).
- В необязательном параметре *op* передается бинарный предикат, определяющий критерий сортировки: *op(elem1, elem2)*.
- Алгоритмы не изменяют состояние исходных интервалов.
- Перед вызовом интервалы должны быть упорядочены по соответствующему критерию сортировки.
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать операторы вставки.
- Приемный интервал не должен перекрываться с исходными интервалами.
- Чтобы без удаления элементов включить в приемный интервал все элементы, присутствующие в обоих исходных интервалах, используйте алгоритм *merge()* (см. с. 406).
- Сложность линейная (не более  $2 \times (\text{numberOfElements1} + \text{numberOfElements2}) - 1$  сравнений).

Пример использования алгоритма *set\_union()* приведен на с. 411. В этом примере также продемонстрированы отличия алгоритма *set\_union()* от других алгоритмов, комбинирующих элементы двух упорядоченных интервалов.

## Пересечение двух упорядоченных множеств элементов

```
OutputIterator
set_intersection (InputIterator source1Beg, InputIterator source1End,
 InputIterator source2Beg, InputIterator source2End,
 OutputIterator destBeg)

set_intersection (InputIterator source1Beg, InputIterator source1End,
 InputIterator source2Beg, InputIterator source2End,
 OutputIterator destBeg, BinaryPredicate op)
```

- Обе формы комбинируют элементы упорядоченных исходных интервалов  $[\text{source1Beg}, \text{source1End}]$  и  $[\text{source2Beg}, \text{source2End}]$  таким образом, что приемный интервал  $[\text{destBeg}, \dots)$  содержит все элементы, присутствующие в обоих интервалах сразу. Например, рассмотрим два интервала:

```
1 2 2 4 6 7 7 9
2 2 2 3 6 6 8 9
```

В результате вызова алгоритма *set\_intersection()* для этих интервалов будет получен следующий интервал:

```
2 2 6 9
```

- В приемном интервале элементы следуют в порядке сортировки.
- Приемный интервал может содержать дубликаты, если соответствующий элемент дублируется в одном из исходных интервалов. Количество элементов с одинаковыми значениями в приемном интервале равно их минимальному количеству в двух исходных интервалах.
- Обе формы возвращают позицию за последним скопированным элементом в приемном интервале.
- В необязательном параметре *op* передается бинарный предикат, определяющий критерий сортировки: *op(elem1, elem2)*.
- Алгоритмы не изменяют состояние исходных интервалов.
- Перед вызовом интервалы должны быть упорядочены по соответствующему критерию сортировки.
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Сложность линейная (не более  $2 \times (\text{numberOfElements1} + \text{numberOfElements2}) - 1$  сравнений).

Пример использования алгоритма `set_intersection()` приведен на с. 411. В этом примере также продемонстрированы отличия алгоритма `set_intersection()` от других алгоритмов, комбинирующих элементы двух упорядоченных интервалов.

## Разность двух упорядоченных множеств элементов

`OutputIterator`

```
set_difference (InputIterator source1Beg, InputIterator source1End,
 InputIterator source2Beg, InputIterator source2End,
 OutputIterator destBeg)
```

```
set_difference (InputIterator source1Beg, InputIterator source1End,
 InputIterator source2Beg, InputIterator source2End,
 OutputIterator destBeg, BinaryPredicate op)
```

- Обе формы комбинируют элементы упорядоченных исходных интервалов `[source1Beg, source1End]` и `[source2Beg, source2End]` таким образом, что приемный интервал `[destBeg, ...)` содержит все элементы, присутствующие в первом интервале, но не входящие во второй интервал. Например, рассмотрим два интервала:

```
1 2 2 4 6 7 7 9
2 2 2 3 6 6 8 9
```

В результате вызова алгоритма `set_difference()` для этих интервалов будет получен следующий интервал:

```
1 4 4 7
```

- В приемном интервале элементы следуют в порядке сортировки.

- Приемный интервал может содержать дубликаты, если соответствующий элемент дублируется в первом исходном интервале. Количество элементов с одинаковыми значениями в приемном интервале равно разности между их количествами в первом и втором исходном интервалах. Если второй интервал содержит больше дубликатов, то количество дубликатов в приемном интервале равно нулю.
- Обе формы возвращают позицию за последним скопированным элементом в приемном интервале.
- В необязательном параметре *op* передается бинарный предикат, определяющий критерий сортировки: *op(elem1, elem2)*.
- Алгоритмы не изменяют состояние исходных интервалов.
- Перед вызовом интервалы должны быть упорядочены по соответствующему критерию сортировки.
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать операторы вставки.
- Приемный интервал не должен перекрываться с исходными интервалами.
- Сложность линейная (не более  $2 \times (\text{numberOfElements1} + \text{numberOfElements2}) - 1$  сравнений).

Пример использования алгоритма `set_difference()` приведен на с. 411. В этом примере также продемонстрированы отличия алгоритма `set_difference()` от других алгоритмов, комбинирующих элементы двух упорядоченных интервалов.

```
OutputIterator
set_symmetric_difference (InputIterator source1Beg,
 InputIterator source1End,
 InputIterator source2Beg,
 InputIterator source2End,
 OutputIterator destBeg)

set_symmetric_difference (InputIterator source1Beg,
 InputIterator source1End,
 InputIterator source2Beg,
 InputIterator source2End,
 OutputIterator destBeg, BinaryPredicate op)
```

- Обе формы комбинируют элементы упорядоченных исходных интервалов  $[source1Beg, source1End]$  и  $[source2Beg, source2End]$  таким образом, что приемный интервал  $[destBeg, ...)$  содержит все элементы, присутствующие либо в первом, либо во втором интервале (но не в обоих сразу). Например, рассмотрим два интервала:

```
1 2 2 4 6 7 7 9
2 2 2 3 6 6 8 9
```

В результате вызова алгоритма `set_symmetric_difference()` для этих интервалов будет получен следующий интервал:

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

- В приемном интервале элементы следуют в порядке сортировки.
- Приемный интервал может содержать дубликаты, если соответствующий элемент дублируется в первом исходном интервале. Количество элементов с одинаковыми значениями в приемном интервале равно разности между их количествами в исходных интервалах.
- Обе формы возвращают позицию за последним скопированным элементом в приемном интервале.
- В необязательном параметре *op* передается бинарный предикат, определяющий критерий сортировки: *op(elem1, elem2)*.
- Алгоритмы не изменяют состояние исходных интервалов.
- Перед вызовом интервалы должны быть упорядочены по соответствующему критерию сортировки.
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать операторы вставки.
- Приемный интервал не должен перекрываться с исходными интервалами.
- Сложность линейная (не более  $2 \times (\text{numberOfElements1} + \text{numberOfElements2}) - 1$  сравнений).

Пример использования алгоритма *set\_difference()* приведен далее. В этом примере также продемонстрированы отличия алгоритма *set\_difference()* от других алгоритмов, комбинирующих элементы двух упорядоченных интервалов.

## Пример использования алгоритмов слияния

В следующем примере сравниваются алгоритмы, комбинирующие элементы двух упорядоченных исходных интервалов, демонстрируются принципы их работы и различия между ними:

```
// algo/setalgos.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
 int c1[] = { 1, 2, 2, 4, 6, 7, 7, 9 };
 int num1 = sizeof(c1) / sizeof(int);

 int c2[] = { 2, 2, 2, 3, 6, 6, 8, 9 };
 int num2 = sizeof(c2) / sizeof(int);

 // Вывод исходных интервалов
 cout << "c1: " ;
 copy (c1, c1+num1,
 ostream_iterator<int>(cout, " "));
 cout << endl;
 cout << "c2: " ;
 copy (c2, c2+num2,
 ostream_iterator<int>(cout, " "));
```

```
cout << '\n' << endl;

// Суммирование интервалов алгоритмом merge()
cout << "merge(): ";
merge (c1, c1+num1,
 c2, c2+num2,
 ostream_iterator<int>(cout, " "));
cout << endl;

// Объединение интервалов алгоритмом set_union()
cout << "set_union(): ";
set_union (c1, c1+num1,
 c2, c2+num2,
 ostream_iterator<int>(cout, " "));
cout << endl;

// Пересечение интервалов алгоритмом set_intersection()
cout << "set_intersection(): ";
set_intersection (c1, c1+num1,
 c2, c2+num2,
 ostream_iterator<int>(cout, " "));
cout << endl;

// Определение элементов первого интервала,
// не входящих во второй интервал, алгоритмом set_difference()
cout << "set_difference(): ";
set_difference (c1, c1+num1,
 c2, c2+num2,
 ostream_iterator<int>(cout, " "));
cout << endl;

// Вычисление разности интервалов алгоритмом set_symmetric_difference()
cout << "set_symmetric_difference(): ";
set_symmetric_difference (c1, c1+num1,
 c2, c2+num2,
 ostream_iterator<int>(cout, " "));
cout << endl;
}
```

Результат выполнения программы выглядит так:

```
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 5 5 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 интервалов

```
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 интервалов  $[beg1, end1beg2]$  и  $[end1beg2, end2]$  так, чтобы интервал  $[beg1, end2]$  содержал упорядоченную совокупность элементов обоих интервалов.
- Сложность линейная при наличии дополнительной памяти ( $numberOfElements - 1$  сравнений),  $n \times \log(n)$  в противном случае ( $numberOfElements \times \log(numberOfElements)$  сравнений).

Пример использования алгоритма `inplace_merge()`:

```
// algo/imergel.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
 list<int> coll;

 // Вставка двух упорядоченных интервалов
 INSERT_ELEMENTS(coll, 1, 7);
 INSERT_ELEMENTS(coll, 1, 8);
 PRINT_ELEMENTS(coll);

 // Определение начала второго интервала (элемент после 7)
 list<int>::iterator pos;
 pos = find (coll.begin(), coll.end(), 7); // Интервал
 // Значение
 ++pos;

 // Слияние в один упорядоченный интервал
 inplace_merge (coll.begin(), pos, coll.end());

 PRINT_ELEMENTS(coll);
}
```

Результат выполнения программы выглядит так:

```
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
```

## Численные алгоритмы

В этом разделе представлены алгоритмы STL, предназначенные для обработки числовых данных. Впрочем, с их помощью можно обрабатывать и другую информацию. Например, алгоритм `accumulate()` может применяться для конкатенации строк. Чтобы использовать числовые алгоритмы, в программу необходимо включить заголовочный файл `<numeric>`<sup>1</sup>:

```
#include <numeric>
```

### Обработка интервалов

#### Вычисление результата по одному интервалу

Т

```
accumulate (InputIterator beg, InputIterator end,
 T initialValue)
```

Т

```
accumulate (InputIterator beg, InputIterator end,
 T initialValue, BinaryFunc op)
```

- Первая форма вычисляет и возвращает сумму `initialValue` и всех элементов в интервале  $[beg, end]$ . Для каждого элемента выполняется команда:

$$\text{initialValue} = \text{initialValue} + \text{elem}$$

- Вторая форма вычисляет и возвращает накопленный результат вызова `op` для `initialValue` и каждого элемента в интервале  $[beg, end]$ . Для каждого элемента выполняется команда:

$$\text{initialValue} = \text{op}(\text{initialValue}, \text{elem})$$

- Таким образом, пусть мы имеем следующие значения:

a1 a2 a3 a4...

Для этих значений соответственно вычисляются и записываются такие величины:

$$\text{initialValue} + a1 + a2 + a3 + \dots$$

$$\text{initialValue} \text{ op } a1 \text{ op } a2 \text{ op } a3 \text{ op } \dots$$

- Для пустого интервала ( $beg == end$ ) обе формы возвращают `initialValue`.
- Предикат `op` не должен модифицировать передаваемые аргументы.
- Сложность линейная (`numberOfElements` вызовов оператора `+` или `op()` соответственно).

---

<sup>1</sup> В исходной версии STL численные алгоритмы определялись в заголовочном файле `<algo.h>`.

В следующем примере алгоритм **accumulate()** используется для вычисления суммы и произведения всех элементов интервала.

```
// algo/accu1.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
 vector<int> coll;

 INSERT_ELEMENTS(coll,1,9);
 PRINT_ELEMENTS(coll);

 // Вычисление суммы элементов
 cout << "sum: "
 << accumulate (coll.begin(), coll.end(),
 0) // Интервал
 // Начальное значение
 << endl;

 // Вычисление суммы элементов с вычетом 100
 cout << "sum: "
 << accumulate (coll.begin(), coll.end(),
 -100) // Интервал
 // Начальное значение
 << endl;

 // Вычисление произведения элементов
 cout << "product: "
 << accumulate (coll.begin(), coll.end(),
 1,
 multiplies<int>()) // Интервал
 // Начальное значение
 // Операция
 << endl;

 // Вычисление произведения элементов (с начальным значением 0)
 cout << "product: "
 << accumulate (coll.begin(), coll.end(),
 0,
 multiplies<int>()) // Интервал
 // Начальное значение
 // Операция
 << endl;
}
```

Результат выполнения программы выглядит так:

```
1 2 3 4 5 6 7 8 9
sum: 45
sum: -55
product: 362880
product: 0
```

Последнее произведение равно нулю, поскольку при умножении любого числа на ноль результат равен нулю.

## Вычисление скалярного произведения двух интервалов

Т

```
inner_product (InputIterator1 beg1, InputIterator1 end1,
 InputIterator2 beg2, T initValue)
```

Т

```
inner_product (InputIterator1 beg1, InputIterator1 end1,
 InputIterator2 beg2, T initValue,
 BinaryFunc op1, BinaryFunc op2)
```

- Первая форма вычисляет и возвращает скалярное произведение *initValue* и элементов интервалов [*beg1*,*end1*) и [*beg2*,...). Для каждого элемента выполняется команда:

$$\text{initValue} = \text{initValue} + \text{elem1} * \text{elem2}$$

- Вторая форма вычисляет и возвращает накопленный результат вызова *op* для *initValue* и элементов интервала [*beg1*,*end1*), объединенных с элементами [*beg2*,...). Для каждого элемента выполняется команда:

$$\text{initValue} = \text{op1}(\text{initValue}, \text{op2}(\text{elem1}, \text{elem2}))$$

- Таким образом, пусть мы имеем следующие значения:

a1 a2 a3 a4...  
b1 b2 b3 ...

Для этих значений соответственно вычисляются и записываются такие величины:

$$\begin{aligned} \text{initValue} &+ (\text{a1} * \text{b1}) + (\text{a2} * \text{b2}) + (\text{a3} * \text{b3}) + \dots \\ \text{initValue} &\text{ op1 (a1 op2 b1) op1 (a2 op2 b2) op1 (a3 op2 b3) op1 } \dots \end{aligned}$$

- Для пустого первого интервала (*beg1* == *end1*) обе формы возвращают *initValue*.
- Перед вызовом необходимо убедиться в том, что интервал [*beg2*,...) содержит достаточно большое количество элементов.
- Предикаты *op1* и *op2* не должны модифицировать передаваемые аргументы.
- Сложность линейная (*numberOfElements* вызовов операторов + и \* или *numberOfElements* вызовов *op1()* и *op2()* соответственно).

В следующем примере алгоритм *inner\_product()* используется для вычисления суммы произведений и произведения сумм элементов двух интервалов.

```
// algo/inner1.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
 list<int> coll;
 INSERT_ELEMENTS(coll, 1, 6);
```

```

PRINT_ELEMENTS(coll);

/* Вычисление суммы произведений элементов
 * (0 + 1*1 + 2*2 + 3*3 + 4*4 + 5*5 + 6*6)
 */
cout << "inner product: "
 << inner_product (coll.begin(), coll.end(), // Первый интервал
 coll.begin(), // Второй интервал
 0) // Начальное значение
 << endl;

/* Вычисление суммы 1*6 ... 6*1
 * (0 + 1*6 + 2*5 + 3*4 + 4*3 + 5*2 + 6*1)
 */
cout << "inner reverse product: "
 << inner_product (coll.begin(), coll.end(), // Первый интервал
 coll.rbegin(), // Второй интервал
 0) // Начальное значение
 << endl;

/* Вычисление произведения сумм элементов
 * (1 * 1+1 * 2+2 * 3+3 * 4+4 * 5+5 * 6+6)
 */
cout << "product of sums: "
 << inner_product (coll.begin(), coll.end(), // Первый интервал
 coll.begin(), // Второй интервал
 1, // Начальное значение
 multiplies<int>(), // Внутренняя операция
 plus<int>()) // Внешняя операция
 << endl;
}

```

Результат выполнения программы выглядит так:

```

1 2 3 4 5 6
inner product: 91
inner reverse product: 56
product od sums: 46080

```

## Преобразования относительных и абсолютных значений

Следующие два алгоритма преобразуют серию относительных значений в серию 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 в интервале  $[sourceBeg, sourceEnd)$  и записывает результат в приемный интервал  $[destBeg, \dots)$ .
- Вторая форма вызывает *op* для каждого элемента в интервале  $[sourceBeg, sourceEnd)$ , объединяет полученное значение со всеми предыдущими значениями и записывает результат в приемный интервал  $[destBeg, \dots)$ .
- Таким образом, пусть мы имеем следующие значения:

$a_1 \ a_2 \ a_3 \ \dots$

Для этих значений соответственно вычисляются и записываются такие величины:

$a_1, \ a_1 + a_2, \ a_1 + a_2 + a_3, \ \dots$   
 $a_1, \ a_1 \text{ op } a_2, \ a_1 \text{ op } a_2 \text{ op } a_3, \ \dots$

- Обе формы возвращают позицию за последним записанным элементом в приемном интервале (то есть позицию первого элемента, который не был заменен).
- Первая форма эквивалентна преобразованию серии относительных значений в серию абсолютных значений. В этом отношении алгоритм `partial_sum()` является логическим дополнением алгоритма `adjacent_difference()`.
- Исходный и приемный интервалы могут быть идентичными.
- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Сложность линейная (*numberOfElements* вызовов оператора `+` или *op()* соответственно).

Пример использования алгоритма `partial_sum()`:

```
// algo/partsum1.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
 vector<int> coll;

 INSERT_ELEMENTS(coll, 1, 6);
 PRINT_ELEMENTS(coll);

 // Вывод всех частичных сумм
 partial_sum (coll.begin(), coll.end(), // Источник
 ostream_iterator<int>(cout, " ")); // Приемник
 cout << endl;
```

```

 // Вывод всех частичных произведений
 partial_sum (coll.begin(), coll.end(), // Источник
 ostream_iterator<int>(cout, " "), // Приемник
 multiplies<int>()); // Операция
 cout << endl;
}

```

Результат выполнения программы выглядит так:

```

1 2 3 4 5 6
1 3 6 10 15 21
1 2 6 24 120 720

```

Пример преобразования относительных значений в абсолютные и наоборот приведен также на с. 420.

## 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 в интервале  $[sourceBeg, sourceEnd)$  и его предшественником и записывает результат в приемный интервал  $[destBeg, \dots)$ .
- Вторая форма вызывает  $op$  для каждого элемента в интервале  $[sourceBeg, sourceEnd)$  и его предшественника и записывает результат в приемный интервал  $[destBeg, \dots)$ .
- Первый элемент просто копируется.
- Таким образом, пусть мы имеем следующие значения:

$a_1 \ a_2 \ a_3 \ a_4 \ \dots$

Для этих значений соответственно вычисляются и записываются такие величины:

$a_1, a_2 - a_1, a_3 - a_2, a_4 - a_3, \dots$   
 $a_1, a_2 \ op \ a_1, a_3 \ op \ a_2, a_4 \ op \ a_3 \dots$

- Обе формы возвращают позицию за последним записанным элементом в приемном интервале (то есть позицию первого элемента, который не был заменен).
- Первая форма эквивалентна преобразованию серии абсолютных значений в серию относительных значений. В этом отношении алгоритм `adjacent_difference()` является логическим дополнением алгоритма `partial_sum()`.
- Исходный и приемный интервалы могут быть идентичными.

- Перед вызовом необходимо убедиться в том, что приемный интервал имеет достаточный размер, или использовать итераторы вставки.
- Предикат *op* не должен модифицировать передаваемые аргументы.
- Сложность линейная (*numberOfElements*-1 вызовов оператора - или *op()* соответственно).

Пример использования алгоритма *adjacent\_difference()*:

```
// algo/adjdiff1.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
 deque<int> coll;
 INSERT_ELEMENTS(coll,1,6);
 PRINT_ELEMENTS(coll);

 // Вывод разностей между элементами
 adjacent_difference (coll.begin(), coll.end(), // Источник
 ostream_iterator<int>(cout, " ")); // Приемник
 cout << endl;

 // Вывод сумм элементов с их предшественниками
 adjacent_difference (coll.begin(), coll.end(), // Источник
 ostream_iterator<int>(cout, " "), // Приемник
 plus<int>()); // Операция
 cout << endl;

 // Вывод произведения элемента и его предшественника
 adjacent_difference (coll.begin(), coll.end(), // Источник
 ostream_iterator<int>(cout, " "), // Приемник
 multiplies<int>()); // Операция
 cout << endl;
}
```

Результат выполнения программы выглядит так:

```
1 2 3 4 5 6
1 1 1 1 1 1
1 3 5 7 9 11
1 2 6 12 20 30
```

Далее приведен еще один пример преобразования относительных значений в абсолютные.

### **Пример преобразования относительных значений в абсолютные**

Следующий пример показывает, как при помощи алгоритмов *partial\_sum()* и *adjacent\_difference()* преобразовать серию относительных значений в серию абсолютных значений и наоборот:

```
// algo/re1abs.cpp
#include "algostuff.hpp"
using namespace std;

int main()
{
 vector<int> coll;

 coll.push_back(17);
 coll.push_back(-3);
 coll.push_back(22);
 coll.push_back(13);
 coll.push_back(13);
 coll.push_back(-9);
 PRINT_ELEMENTS(coll, "coll: ");

 // Преобразование к относительным значениям
 adjacent_difference (coll.begin(), coll.end(), // Источник
 coll.begin()); // Приемник
 PRINT_ELEMENTS(coll, "relative: ");

 // Преобразование к абсолютным значениям
 partial_sum (coll.begin(), coll.end(), // Источник
 coll.begin()); // Приемник
 PRINT_ELEMENTS(coll, "absolute: ");
}
```

Результат выполнения программы выглядит так:

```
coll: 17 -3 22 13 13 -9
relative: 17 -20 25 -9 0 -22
absolute: 17 -3 22 13 13 -9
```

# 10 Специальные контейнеры

Стандартная библиотека C++ не ограничивается контейнерами, входящими в STL. В нее также включены контейнеры, предназначенные для особых целей и обладающие простыми, почти очевидными интерфейсами. Такие контейнеры можно разделить на группы.

## ○ Так называемые *контейнерные адаптеры*.

К этой группе относятся контейнеры, адаптирующие стандартные контейнеры STL для особых целей. Существуют три стандартных контейнерных адаптера:

- стеки;
- очереди;
- приоритетные очереди.

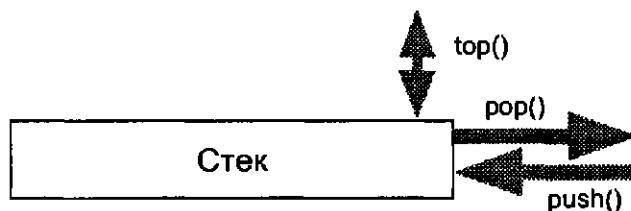
*Приоритетными очередями* называются очереди, элементы которых автоматически сортируются в соответствии с заданным критерием сортировки. Таким образом, значение «следующего» элемента приоритетной очереди «больше» значения «предыдущего».

## ○ Специальный контейнер *bitset*.

Контейнер *bitset* представляет собой битовое поле с произвольным, но фиксированным количеством битов. В стандартную библиотеку C++ также включен специальный контейнер переменного размера для логических значений *vector<bool>* (см. с. 167).

## Стеки

Класс *stack<>* реализует стек, работающий по принципу «последним прибыл, первым обслужен» (LIFO). Функция *push()* вставляет в стек произвольное количество элементов (рис. 10.1), а функция *pop()* удаляет элементы в порядке, обратном порядку их вставки.



**Рис. 10.1.** Интерфейс стека

Чтобы использовать стек в программе, необходимо включить в нее заголовочный файл `<stack>`<sup>1</sup>:

```
#include <stack>
```

В файле `<stack>` класс `stack` определяется следующим образом:

```
namespace std {
 template <class T,
 class Container = deque<T> >
 class stack;
}
```

Первый параметр шаблона определяет тип элементов. Необязательный второй параметр шаблона определяет контейнер, который будет использоваться внутренней реализацией для хранения элементов. По умолчанию это дек. Такой выбор объясняется тем, что деки (в отличие от векторов) освобождают память при удалении элементов и обходятся без копирования всех элементов при перераспределении памяти (рекомендации относительно того, когда следует выбирать тот или иной контейнер, приведены на с. 229).

Например, следующее объявление определяет стек с элементами типа `int`<sup>2</sup>:

```
std::stack<int> st; // Стек целых чисел
```

Реализация стека просто отображает операции со стеком на соответствующие операции с используемым контейнером (рис. 10.2). Допускается применение любого класса последовательного контейнера с поддержкой функций `back()`, `push_back()` и `pop_back()`. Например, элементы стека могут храниться в векторе или списке:

```
std::stack<int> st; // Стек целых чисел на базе вектора
```

## Основной интерфейс

Основной интерфейс стеков состоит из функций `push()`, `top()` и `pop()`:

- функция `push()` вставляет элемент в стек;
- функция `top()` возвращает верхний элемент стека;
- функция `pop()` удаляет элемент из стека.

<sup>1</sup> В исходной версии STL стек определялся в заголовочном файле `<stack.h>`.

<sup>2</sup> В предыдущих версиях STL единственным параметром шаблона был тип контейнера, а объявление стека с элементами типа `int` выглядело так:

```
stack<deque<int> > st;
```

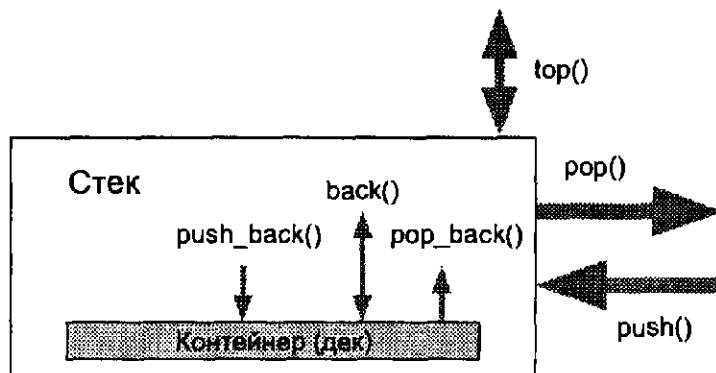


Рис. 10.2. Внутренний интерфейс стека

Обратите внимание: функция `pop()` удаляет верхний элемент, но не возвращает его, тогда как функция `top()` возвращает верхний элемент стека без удаления. Следовательно, чтобы обработать верхний элемент и удалить его из стека, всегда приходится вызывать обе функции. Такой интерфейс несколько неудобен, но он более эффективен при удалении верхнего элемента без его обработки. Если стек не содержит ни одного элемента, поведение функций `top()` и `pop()` не определено. Наличие элементов в стеке проверяется функциями `size()` и `empty()`.

Если стандартный интерфейс `stack<>` вас не устраивает, вы легко можете написать более удобный интерфейс. Пример приведен на с. 428.

## Пример использования стека

Пример использования класса `stack<>`:

```

// cont/stack1.cpp
#include <iostream>
#include <stack>
using namespace std;

int main()
{
 stack<int> st;

 // Занесение трех элементов в стек
 st.push(1);
 st.push(2);
 st.push(3);

 // Извлечение и вывод двух элементов из стека
 cout << st.top() << ' ';
 st.pop();
 cout << st.top() << ' ';
 st.pop();

 // Модификация верхнего элемента
 st.top() = 77;

 // Занесение двух новых элементов

```

```

st.push(4);
st.push(5);

// Извлечение одного элемента без обработки
st.pop();

// Извлечение и вывод оставшихся элементов
while (!st.empty()) {
 cout << st.top() << ' ';
 st.pop();
}
cout << endl;
}

```

Результат выполнения программы:

3 2 4 77

## Строение класса stack

Интерфейс класса `stack<T>` настолько компактен, что в нем можно легко разобраться, проанализировав типичную реализацию:

```

namespace std {
 template <class T, class Container = deque<T> >
 class stack {
 public:
 typedef typename Container::value_type value_type;
 typedef typename Container::size_type size_type;
 typedef Container container_type;
 protected:
 Container c; // Контейнер
 public:
 explicit stack(const Container& = Container());
 bool empty() const { return c.empty(); }
 size_type size() const { return c.size(); }
 void push(const value_type& x) { c.push_back(x); }
 void pop() { c.pop_back(); }
 value_type& top() { return c.back(); }
 const value_type& top() const { return c.back(); }
 };
 template <class T, class Container>
 bool operator==(const stack<T, Container>&, const stack<T, Container>&);
 template <class T, class Container>
 bool operator< (const stack<T, Container>&, const stack<T, Container>&);
 ... // (Другие операторы сравнения)
 }
}

```

Ниже приводятся более подробные описания членов класса `stack<T>`.

## Определения типов

`стек::value_type`

- Тип элементов.
- Эквивалент:

`контейнер::value_type`

`стек::size_type`

- Беззнаковый целый тип для значений размера.
- Эквивалент:

`контейнер::size_type`

`стек::container_type`

Тип контейнера.

## Операции

`стек::stack ()`

- Конструктор по умолчанию.
- Создает пустой стек.

`explicit стек::stack (const Container& cont)`

- Создает стек, инициализированный элементами *cont*.
- Все элементы *cont* копируются в стек.

`size_type стек::size () const`

- Возвращает текущее количество элементов.
- Для проверки отсутствия элементов в стеке рекомендуется использовать функцию `empty()`, поскольку она может работать быстрее.

`bool стек::empty () const`

- Проверяет, пуст ли стек.
- Эквивалент (но может работать быстрее):

`стек::size()==0`

`void стек::push (const value_type& elem)`

- Вставляет копию *elem* в стек, в результате чего она становится новым первым элементом.

`value_type& стек::top ()`

`const value_type& стек::top () const`

- Обе формы возвращают верхний элемент стека, то есть элемент, который был вставлен последним (после всех остальных элементов стека).

- Перед вызовом необходимо убедиться в том, что стек содержит хотя бы один элемент (`size()>0`), иначе вызов приводит к непредсказуемым последствиям.
- Первая форма для неконстантных стеков возвращает ссылку, что позволяет модифицировать верхний элемент во время его нахождения в стеке. Хорошо это или нет — решайте сами.

```
void стек::pop ()
```

- Удаляет из стека верхний элемент, то есть элемент, который был вставлен последним (после всех остальных элементов стека).
- Функция не имеет возвращаемого значения. Чтобы обработать значение верхнего элемента, следует предварительно вызвать функцию `top()`.
- Перед вызовом необходимо убедиться в том, что стек содержит хотя бы один элемент (`size()>0`), иначе вызов приводит к непредсказуемым последствиям.

```
bool сравнение (const стек& stack1, const стек& stack2)
```

- Возвращает результат сравнения двух стеков одного типа.
- Параметр *сравнение* — одна из следующих операций:  
`operator ==`  
`operator !=`  
`operator <`  
`operator >`  
`operator <=`  
`operator >=`
- Два стека считаются равными, если они содержат одинаковое количество элементов, если элементы попарно совпадают и следуют в одинаковом порядке (то есть проверка двух соответствующих друг другу элементов на равенство всегда дает `true`).
- Отношение «меньше/больше» между контейнерами проверяется по лексикографическому критерию. Лексикографический критерий рассматривается при описании алгоритма `lexicographical_compare()` на с. 356.

## Пользовательская реализация стека

Стандартный класс `stack<>` ставит на первое место не удобство и безопасность, а скорость работы. У автора на этот счет другое мнение, поэтому вашему вниманию предлагается нестандартная реализация стека. Она обладает двумя преимуществами:

- функция `pop()` возвращает верхний элемент;
- при пустом стеке функции `pop()` и `top()` генерируют исключения.

Кроме того, из реализации были исключены операции, лишние для рядового пользователя стеков (например, операции сравнения). Полученный класс стека приведен ниже.

```
// cont/Stack.hpp
/* ****
 * Stack.hpp
 * - более удобный и безопасный класс стека
 * ****/
#ifndef STACK_HPP
#define STACK_HPP

#include <deque>
#include <exception>

template <class T>
class Stack {
protected:
 std::deque<T> c; // Контейнер для хранения элементов

public:
 /* Класс исключения, генерируемого функциями 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();
 }

 // Проверка пустого стека
 bool empty() const {
 return c.empty();
 }

 // Занесение элемента в стек
 void push (const T& elem) {
 c.push_back(elem);
 }

 // Извлечение элемента из стека с возвращением его значения
 T pop () {
 if (c.empty()) {
 throw ReadEmptyStack();
 }
 }
}
```

```
 T elem(c.back());
 c.pop_back();
 return elem;
}

// Получение значения верхнего элемента
T& top () {
 if (c.empty()) {
 throw ReadEmptyStack();
 }
 return c.back();
}
};

#endif /* STACK_HPP */
```

При использовании этого класса стека предыдущий пример можно записать в следующем виде:

```
// cont/stack2.cpp
#include <iostream>
#include "Stack.hpp" // Использование нестандартного класса стека
using namespace std;

int main()
{
 try {
 Stack<int> st;

 // Занесение трех элементов в стек
 st.push(1);
 st.push(2);
 st.push(3);

 // Извлечение и вывод двух элементов из стека
 cout << st.pop() << ' ';
 cout << st.pop() << ' ';

 // Модификация верхнего элемента
 st.top() = 77;

 // Занесение двух новых элементов
 st.push(4);
 st.push(5);

 // Извлечение одного элемента без обработки
 st.pop();

 /* Извлечение и вывод трех элементов
 * - ОШИБКА: одного элемента не хватает
```

```

 */
cout << st.pop() << ' ';
cout << st.pop() << endl;
cout << st.pop() << endl;
}
catch (const exception& e) {
 cerr << "EXCEPTION: " << e.what() << endl;
}
}

```

При последнем вызове `pop()` намеренно допущена ошибка. В отличие от стандартного класса стека с неопределенным поведением наш класс генерирует исключение. Результат выполнения программы выглядит так:

```

3 2 4 77
EXCEPTION: read empty stack

```

## Очереди

Класс `queue<>` реализует очередь, работающую по принципу «первым поступил, первым обслужен» (FIFO). Функция `push()` заносит элементы в очередь (рис. 10.3), а функция `pop()` удаляет элементы в порядке их вставки. Следовательно, очередь может рассматриваться как классический буфер данных.



Рис. 10.3. Интерфейс очереди

Чтобы использовать очередь в программе, необходимо включить заголовочный файл `<queue>`<sup>1</sup>:

```
#include <queue>
```

В файле `<queue>` класс `queue` определяется следующим образом:

```

namespace std {
 template <class T,
 class Container = deque<T> >
 class queue;
}

```

<sup>1</sup> В исходной версии STL очередь определялась в заголовочном файле `<queue.h>`.

Первый параметр шаблона определяет тип элементов. Необязательный второй параметр шаблона определяет контейнер, который будет использоваться внутренней реализацией для хранения элементов. По умолчанию это дек.

Например, следующее объявление определяет очередь со строковыми элементами<sup>1</sup>:

```
std::queue<std::string> buffer; // Строковая очередь
```

Реализация очереди просто отображает операции с очередью на соответствующие операции с используемым контейнером (рис. 10.4). Допускается применение любого класса последовательного контейнера с поддержкой функций `front()`, `back()`, `push_back()` и `pop_front()`. Например, элементы очереди могут храниться в списке:

```
std::queue<std::string, std::list<std::string>> buffer;
```

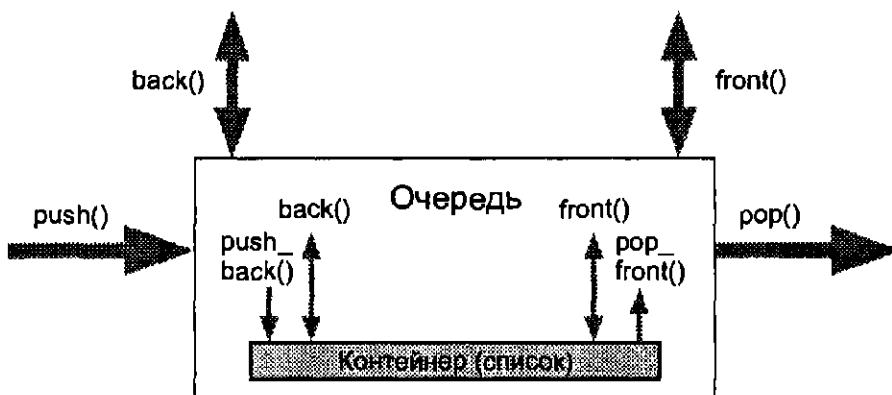


Рис. 10.4. Внутренний интерфейс очереди

## Основной интерфейс

Основной интерфейс очередей состоит из функций `push()`, `front()`, `back()` и `pop()`:

- функция `push()` вставляет элемент в очередь;
- функция `front()` возвращает следующий элемент очереди (тот, который был вставлен раньше других);
- функция `back()` возвращает последний элемент очереди (тот, который был вставлен позже других);
- функция `pop()` удаляет элемент из очереди.

Обратите внимание: функция `pop()` удаляет следующий элемент, но не возвращает его, тогда как функции `front()` и `back()` возвращают следующий элемент без удаления. Следовательно, чтобы обработать следующий элемент и удалить его из очереди, всегда приходится вызывать функции `front()` и `pop()`. Такой интерфейс несколько исудобен, но он более эффективен при удалении следующего элемента без его обработки. Если очередь не содержит ни одного элемента,

<sup>1</sup> В предыдущих версиях STL единственным параметром шаблона был тип контейнера, а объявление очереди с элементами типа `string` выглядело так:  
`queue<deque<string>> buffer;`

поведение функций `front()`, `back()` и `pop()` не определено. Наличие элементов в очереди проверяется функциями `size()` и `empty()`.

Если стандартный интерфейс `queue<>` вас не устраивает, вы легко можете написать более удобный интерфейс. Пример приведен на с. 436.

## Пример использования очереди

Пример использования класса `queue<>`:

```
// cont/queue1.cpp
#include <iostream>
#include <queue>
#include <string>
using namespace std;

int main()
{
 queue<string> q;

 // Вставка трех элементов в очередь
 q.push("These ");
 q.push("are ");
 q.push("more than ");

 // Вывод и удаление двух элементов из очереди
 cout << q.front();
 q.pop();
 cout << q.front();
 q.pop();

 // Вставка двух новых элементов
 q.push("four ");
 q.push("words!");

 // Удаление одного элемента
 q.pop();

 // Вывод и удаление двух элементов
 cout << q.front();
 q.pop();
 cout << q.front() << endl;
 q.pop();

 // Вывод количества элементов в очереди
 cout << "number of elements in the queue: " << q.size()
 << endl;
}
```

Результат выполнения программы выглядит так:

```
These are four words!
number of elements in the queue: 0
```

## Строение класса `queue`

Типичная реализация класса `queue<T>`, как и реализация класса `stack<T>`, не требует особых комментариев:

```
namespace std {
 template <class T, class Container = deque<T> >
 class queue {
 public:
 typedef typename Container::value_type value_type;
 typedef typename Container::size_type size_type;
 typedef Container container_type;
 protected:
 Container c; // Контейнер
 public:
 explicit queue(const Container& = Container());
 bool empty() const { return c.empty(); }
 size_type size() const { return c.size(); }
 void push(const value_type& x) { c.push_back(x); }
 void pop() { c.pop_front(); }
 value_type& front() { return c.front(); }
 const value_type& front() const { return c.front(); }
 value_type& back() { return c.back(); }
 const value_type& back() const { return c.back(); }
 };
 template <class T, class Container>
 bool operator==(const queue<T, Container>&,
 const queue<T, Container>&);
 template <class T, class Container>
 bool operator< (const queue<T, Container>&,
 const queue<T, Container>&);
 ... // (Другие операторы сравнений)
 }
}
```

Ниже приводятся более подробные описания членов класса `queue<T>`.

### Определения типов

`очередь::value_type`

○ Тип элементов.

○ Эквивалент:

`контейнер::value_type`

`очередь::size_type`

- Беззнаковый целый тип для значений размера.
- Эквивалент:

`контейнер::size_type`

`очередь::container_type`

Тип контейнера.

## Операции

`очередь::queue ()`

- Конструктор по умолчанию.
- Создает пустую очередь.

`explicit очередь::queue (const Container& cont)`

- Создает очередь, инициализированную элементами *cont*.
- Все элементы *cont* копируются в очередь.

`size_type очередь::size () const`

- Возвращает текущее количество элементов.
- Для проверки отсутствия элементов в очереди рекомендуется использовать функцию `empty()`, поскольку она может работать быстрее.

`bool очередь::empty () const`

- Проверяет, пуста ли очередь.
- Эквивалент (но может работать быстрее):

`очередь::size() == 0`

`void очередь::push (const value_type& elem)`

Вставляет в очередь копию параметра *elem*, в результате чего она становится новым последним элементом.

`value_type& очередь::front ()`

`const value_type& очередь::front () const`

- Обе формы возвращают следующий элемент очереди, то есть элемент, который был вставлен первым (после всех остальных элементов очереди).
- Перед вызовом необходимо убедиться в том, что очередь содержит хотя бы один элемент (`size() > 0`), иначе вызов приводит к непредсказуемым последствиям.
- Первая форма для неконстантных очередей возвращает ссылку, что позволяет модифицировать следующий элемент во время его нахождения в очереди. Хорошо это или нет — решайте сами.

```
value_type& очередь::back ()
const value_type& очередь:: back () const
```

- Обе формы возвращают последний элемент очереди, то есть элемент, который был вставлен последним (после всех остальных элементов очереди).
- Перед вызовом необходимо убедиться в том, что очередь содержит хотя бы один элемент (`size()>0`), иначе вызов приводит к непредсказуемым последствиям.
- Первая форма для неконстантных очередей возвращает ссылку, что позволяет модифицировать последний элемент во время его нахождения в очереди. Хорошо это или нет — решайте сами.

```
void очередь::pop ()
```

- Удаляет из очереди следующий элемент, то есть элемент, который был вставлен первым (перед всеми элементами очереди).
- Функция не имеет возвращаемого значения. Чтобы обработать значение верхнего элемента, следует предварительно вызвать функцию `front()`.
- Перед вызовом необходимо убедиться в том, что очередь содержит хотя бы один элемент (`size()>0`), иначе вызов приводит к непредсказуемым последствиям.

```
bool сравнение (const очередь& queue1, const очередь& queue2)
```

- Возвращает результат сравнения двух очередей одного типа.
- Параметр *сравнение* — одна из следующих операций:

```
operator ==
operator !=
operator <
operator >
operator <=
operator >=
```

- Две очереди считаются равными, если они содержат одинаковое количество элементов, если элементы попарно совпадают и следуют в одинаковом порядке (то есть проверка двух соответствующих друг другу элементов на равенство всегда дает `true`).
- Отношение «меньше/больше» между контейнерами проверяется по лексикографическому критерию. Лексикографический критерий рассматривается при описании алгоритма `lexicographical_compare()` на с. 356.

## Пользовательская реализация очереди

Стандартный класс `queue<>` ставит на первое место не удобство и безопасность, а скорость работы. У автора на этот счет другое мнение, поэтому вашему вниманию предлагается нестандартная реализация очереди. Она обладает двумя преимуществами:

- функция `pop()` возвращает следующий элемент;
- при пустой очереди функции `pop()` и `front()` генерируют исключения.

Кроме того, из реализации исключены операции, лишние для рядового пользователя очередей (например, операции сравнения и функция `back()`). Полученный класс очереди приведен ниже.

```
// cont/queue.hpp
/*
 * Queue.hpp
 * - более удобный и безопасный класс очереди
 */
#ifndef QUEUE_HPP
#define QUEUE_HPP

#include <deque>
#include <exception>

template <class T>
class Queue {
protected:
 std::deque<T> c; // Контейнер для хранения элементов

public:
 /* Класс исключения, генерируемого функциями pop() и front()
 * при пустой очереди
 */
 class ReadEmptyQueue : public std::exception {
 public:
 virtual const char* what() const throw() {
 return "read empty queue";
 }
 };

 // Количество элементов
 typename std::deque<T>::size_type size() const {
 return c.size();
 }

 // Проверка пустой очереди
 bool empty() const {
 return c.empty();
 }

 // Занесение элемента в очередь
 void push (const T& elem) {
 c.push_back(elem);
 }

 // Извлечение элемента из очереди с возвращением его значения
 T pop () {
 if (c.empty()) {
 throw ReadEmptyQueue();
 }
 }
}
```

```
T elem(c.front());
c.pop_front();
return elem;
}

// Получение значения следующего элемента
T& front () {
 if (c.empty()) {
 throw ReadEmptyQueue();
 }
 return c.front();
}
};

#endif /* QUEUE_HPP */
```

При использовании этой очереди предыдущий пример можно записать в следующем виде:

```
// cont/queue2.cpp
#include <iostream>
#include <string>
#include "Queue.hpp" // Использование нестандартного класса очереди
using namespace std;

int main()
{
 try {
 Queue<string> q;

 // Вставка трех элементов в очередь
 q.push("These ");
 q.push("are ");
 q.push("more than ");

 // Вывод и удаление двух элементов из очереди
 cout << q.pop();
 cout << q.pop();

 // Вставка двух новых элементов
 q.push("four ");
 q.push("words!");

 // Удаление одного элемента
 q.pop();

 // Вывод и удаление двух элементов из очереди
 cout << q.pop();
 cout << q.pop() << endl;

 // Вывод количества оставшихся элементов
 cout << "number of elements in the queue: " << q.size()
 << endl;
 }
}
```

```

 // Вывод и удаление одного элемента
 cout << q.pop() << endl;
}
catch (const exception& e) {
 cerr << "EXCEPTION: " << e.what() << endl;
}
}
}

```

При последнем вызове `pop()` намеренно допущена ошибка. В отличие от стандартного класса очереди с неопределенным поведением наш класс генерирует исключение. Результат выполнения программы выглядит так:

```

These are four words!
number of elements in the queue: 0
EXCEPTION: read empty queue

```

## Приоритетные очереди

Класс `priority_queue<>` реализует очередь, в которой последовательность чтения элементов определяется их приоритетами. По своему интерфейсу приоритетные очереди близки к обычным очередям: функция `push()` заносит элемент в очередь, а функции `top()` и `pop()` читают и удаляют следующий элемент (рис. 10.5). Однако в приоритетных очередях следующим элементом является не первый вставленный элемент, а элемент с максимальным приоритетом. Таким образом, можно считать, что порядок сортировки элементов частично определяется их значениями. Как обычно, критерий сортировки может передаваться в параметре шаблона. По умолчанию элементы сортируются оператором `<` в порядке убывания; при этом следующим элементом всегда является элемент с максимальным приоритетом. Если существует несколько элементов с «максимальными» приоритетами, критерий выбора определяется реализацией.

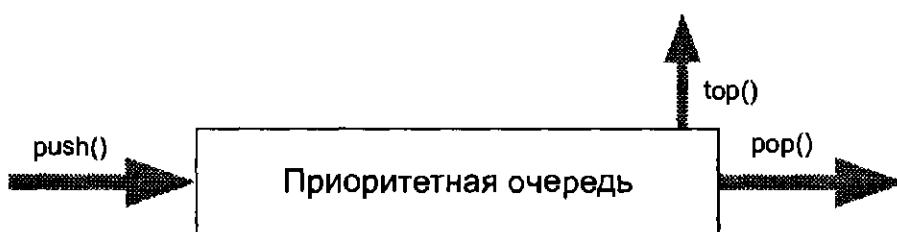


Рис. 10.5. Интерфейс приоритетной очереди

Приоритетные очереди определяются в том же заголовочном файле `<queue>`, в котором определяются обычные очереди<sup>1</sup>:

```
#include <queue>
```

В файле `<queue>` класс `priority_queue` определяется следующим образом:

```
namespace std {
```

<sup>1</sup> В исходной версии STL очередь определялась в заголовочном файле `<queue.h>`.

```
template <class T,
 class Container = vector<T>,
 class Compare = less<typename Container::value_type>>
class priority_queue;
```

Первый параметр шаблона определяет тип элементов. Необязательный второй параметр шаблона определяет контейнер, который будет использоваться внутренней реализацией приоритетной очереди для хранения элементов. По умолчанию это вектор. Необязательный третий параметр шаблона определяет критерий сортировки, который требуется для поиска следующего элемента с высшим приоритетом. По умолчанию элементы сравниваются оператором <.

Например, следующее объявление определяет приоритетную очередь с элементами типа `float`<sup>1</sup>:

```
std::priority_queue<float> pbuffer; // Приоритетная очередь для типа float
```

Реализация приоритетной очереди просто отображает операции с очередью на соответствующие операции с используемым контейнером (см. рис. 10.4). Допускается использование любого класса последовательного контейнера с поддержкой функций итераторов произвольного доступа и функций `front()`, `back()`, `push_back()` и `pop_back()`. Произвольный доступ необходим для сортировки элементов, выполняемой алгоритмами STL для работы с кучей (см. с. 396). Например, элементы приоритетной очереди могут храниться в деке:

```
std::priority_queue<float, std::deque<float> > pbuffer;
```

Чтобы определить собственный критерий сортировки, необходимо передать бинарный предикат в виде функции или объекта функции; предикат используется алгоритмами сортировки для сравнения двух элементов (за информацией о критериях сортировки обращайтесь на с. 186 и 296). Например, следующее объявление определяет приоритетную очередь с обратным порядком сортировки:

```
std::priority_queue<float, std::vector<float>,
 std::greater<float> > pbuffer;
```

В такой приоритетной очереди следующим элементом всегда является один из элементов с наименьшим значением.

## Основной интерфейс

Основной интерфейс приоритетных очередей состоит из функций `push()`, `top()` и `pop()`:

- функция `push()` вставляет элемент в приоритетную очередь;
- функция `top()` возвращает следующий элемент приоритетной очереди;
- функция `pop()` удаляет элемент из приоритетной очереди.

<sup>1</sup> В предыдущих версиях STL шаблону всегда передавался тип контейнера и критерий сортировки, поэтому объявление приоритетной очереди с элементами типа `float` выглядело так: `priority_queue<vector<float>, less<float> > buffer;`

Как и в остальных контейнерных адаптерах, функция `pop()` удаляет следующий элемент, но не возвращает его, тогда как функция `top()` возвращает следующий элемент без удаления. Следовательно, чтобы обработать и удалить следующий элемент очереди, всегда приходится вызывать обе функции. И, как обычно, если очередь не содержит ни одного элемента, поведение функций `top()` и `pop()` не определено. Наличие элементов в очереди проверяется функциями `size()` и `empty()`.

## Пример использования приоритетных очередей

Пример использования класса `priority_queue<>`:

```
// cont/pqueue1.cpp
#include <iostream>
#include <queue>
using namespace std;

int main()
{
 priority_queue<float> q;

 // Вставка трех элементов в приоритетную очередь
 q.push(66.6);
 q.push(22.2);
 q.push(44.4);

 // Вывод и удаление двух элементов
 cout << q.top() << ' ';
 q.pop();
 cout << q.top() << endl;
 q.pop();

 // Вставка еще трех элементов
 q.push(11.1);
 q.push(55.5);
 q.push(33.3);

 // Удаление одного элемента
 q.pop();

 // Извлечение и вывод оставшихся элементов
 while (!q.empty()) {
 cout << q.top() << ' ';
 q.pop();
 }
 cout << endl;
}
```

Результат выполнения программы выглядит так:

```
66.6 44.4
33.3 22.2 11.1
```

Как видите, после вставки значений 66.6, 22.2 и 44.4 программа считает старшими элементами 66.6 и 44.4. После вставки еще трех элементов приоритетная очередь содержит значения 22.2, 11.1, 55.5 и 33.3 (в порядке вставки). Следующий элемент просто удаляется вызовом `pop()`, поэтому итоговый цикл выводит значения 33.3, 22.2 и 11.1 именно в этом порядке.

## Строение класса `priority_queue`

Типичная реализация класса `priority_queue<>`, как и реализаций классов `stack<>` и `queue<>`, понятна без комментариев:

```
namespace std {
 template <class T, class Container = vector<T>,
 class Compare = less<typename Container::value_type> >
 class priority_queue {
 public:
 typedef typename Container::value_type value_type;
 typedef typename Container::size_type size_type;
 typedef Container container_type;
 protected:
 Compare comp; // Критерий сортировки
 Container c; // Контеинер
 public:
 // Конструкторы
 explicit priority_queue(const Compare& cmp = Compare(),
 const Container& = 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(); }
 };
}
```

Как видно из приведенного листинга, приоритетная очередь использует алгоритмы STL для работы с кучей, описанные на с. 396. В отличие от других контейнерных адаптеров для приоритетной очереди не определены операторы сравнения.

Ниже приводятся более подробные описания членов класса `priority_queue<>`.

## Определения типов

`приоритетная_очередь::value_type`

- Тип элементов.
- Эквивалент:

`контейнер::value_type`

`приоритетная_очередь::size_type`

- Беззнаковый целый тип для значений размера.
- Эквивалент:

`контейнер::size_type`

`приоритетная_очередь::container_type`

Тип контейнера.

## Конструкторы

`приоритетная_очередь::priority_queue ()`

- Конструктор по умолчанию.
- Создает пустую приоритетную очередь.

`explicit приоритетная_очередь::priority_queue (const CompFunc& op)`

- Создает пустую приоритетную очередь с критерием сортировки *op*.
- Примеры передачи критерия сортировки в аргументах конструктора приведены на с. 198 и 218.

`приоритетная_очередь::priority_queue (const CompFunc& op,  
const Container& cont)`

- Создает приоритетную очередь с критерием сортировки *op* и инициализирует ее элементами *cont*.
- Данная функция объявлена как шаблонная (см. с. 28), поэтому элементы исходного интервала могут относиться к любому типу, который преобразуется к типу элементов контейнера.

`приоритетная_очередь::priority_queue (InputIterator beg,  
InputIterator end)`

- Создаст приоритетную очередь с критерием сортировки *op* и инициализирует ее элементами интервала *[beg,end)*.
- Данная функция объявлена как шаблонная (см. с. 28), поэтому элементы исходного интервала могут относиться к любому типу, который преобразуется к типу элементов контейнера.

```
приоритетная_очередь::priority_queue (InputIterator beg, InputIterator end,
 const CompFunc& op)
```

- Создает приоритетную очередь с критерием сортировки *op* и инициализирует ее элементами интервала *[beg,end]*.
- Данная функция объявлена как шаблонная (см. с. 28), поэтому элементы исходного интервала могут относиться к любому типу, который преобразуется к типу элементов контейнера.
- Примеры передачи критерия сортировки в аргументах конструктора приведены на с. 198 и 218.

```
приоритетная_очередь::priority_queue (InputIterator beg, InputIterator end,
 const CompFunc& op,
 const Container& cont)
```

- Создает приоритетную очередь с критерием сортировки *op* и инициализирует ее элементами контейнера *cont* и интервала *[beg,end]*.
- Данная функция объявлена как шаблонная (см. с. 28), поэтому элементы исходного интервала могут относиться к любому типу, который преобразуется к типу элементов контейнера.

## Другие операции

```
size_type приоритетная_очередь::size () const
```

- Возвращает текущее количество элементов.
- Для проверки отсутствия элементов в приоритетной очереди рекомендуется использовать функцию *empty()*, поскольку она может работать быстрее.

```
bool приоритетная_очередь::empty () const
```

- Проверяет, пуста ли приоритетная очередь.
- Эквивалент (но может работать быстрее):

```
приоритетная_очередь::size() == 0
```

```
void приоритетная_очередь::push (const value_type& elem)
```

Вставляет копию *elem* в приоритетную очередь.

```
const value_type& приоритетная_очередь::top () const
```

- Возвращает следующий элемент приоритетной очереди (то есть элемент с наибольшим значением среди всех элементов очереди). Если приоритетная очередь содержит сразу несколько элементов с таким значением, критерий выбора определяется реализацией.
- Перед вызовом необходимо убедиться в том, что приоритетная очередь содержит хотя бы один элемент (*size() > 0*), иначе вызов приводит к непредсказуемым последствиям.

```
void приоритетная_очередь::pop ()
```

- Удаляет из приоритетной очереди следующий элемент (то есть элемент с наибольшим значением среди всех элементов очереди). Если приоритетная очередь содержит сразу несколько элементов с таким значением, критерий выбора определяется реализацией.
- Функция не имеет возвращаемого значения. Чтобы обработать значение следующего элемента, следует предварительно вызвать функцию `top()`.
- Перед вызовом необходимо убедиться в том, что приоритетная очередь содержит хотя бы один элемент (`size() > 0`), иначе вызов приводит к непредсказуемым последствиям.

## Битовые поля

Битовые поля моделируют массивы битов (логических величин) фиксированного размера. Они часто используются для выполнения операций с наборами независимых флагов. В программах на языке C, а также в старых программах C++ массив битов обычно представляется типом `long`, а операции с битами выполняются при помощи поразрядных операторов (таких, как `&`, `|` и `~`). Основные достоинства класса `bitset` — произвольный размер битового поля и поддержка дополнительных операций (например, присваивание значений отдельных битов, чтение и запись битовых полей как последовательности нулей и единиц).

Количество битов в битовом поле остается неизменным, поскольку оно передается в параметре шаблона. Если вам потребуется контейнер с переменным количеством битов, воспользуйтесь классом `vector<bool>` (см. с. 167).

Класс `bitset` определяется в заголовочном файле `<bitset>`:

```
#include <bitset>
```

В файле `<bitset>` класс `bitset` определяется в виде шаблона, которому в параметре передается размер поля в битах:

```
namespace std {
 template <size_t Bits>
 class bitset;
}
```

В данном случае в параметре шаблона передается не тип, а беззнаковое целое значение (эта возможность рассматривается на с. 26).

Шаблоны с разными аргументами определяют разные типы. Сравнение и комбинирование битовых полей разрешено только для полей с одинаковыми значениями аргумента шаблона.

## Примеры использования битовых полей

### Битовое поле как набор флагов

В первом примере битовое поле интерпретируется как набор флагов. Каждый флаг представлен некоторым значением перечисляемого типа, определяющим

позицию бита в поле. В нашем примере биты представляют различные цвета; таким образом, каждое значение перечисляемого типа определяет один цвет. Битовые поля являются удобным средством для работы с переменными, содержащими произвольную комбинацию цветов:

```
// cont/bitset1.cpp
#include <bitset>
#include <iostream>
using namespace std;

int main()
{
 /* enumeration type for the bits
 * - each bit represents a color
 */
 enum Color { red, yellow, green, blue, white, black, //...
 numColors };

 // Создание битового поля для всех битов/цветов
 bitset<numColors> usedColors;

 // Установка битов двух цветов
 usedColors.set(red);
 usedColors.set(blue);

 // Вывод данных битового поля
 cout << "bitfield of used colors: " << usedColors
 << endl;
 cout << "number of used colors: " << usedColors.count()
 << endl;
 cout << "bitfield of unused colors: " << ~usedColors
 << endl;

 // Если в битовом поле использован хотя бы один цвет...
 if (usedColors.any()) {
 // перебрать все цвета в цикле
 for (int c = 0; c < numColors; ++c) {
 // Если флаг текущего цвета установлен...
 if (usedColors[(Color)c]) {
 //...
 }
 }
 }
}
```

## Представление двоичных данных при вводе-выводе

Другая полезная возможность битовых полей — преобразование целых чисел в последовательность битов и наоборот. Задача решается простым созданием временного битового поля:

```
// cont/bitset2.cpp
#include <bitset>
#include <iostream>
#include <string>
#include <limits>
using namespace std;

int main()
{
 /* Вывод чисел в двоичном 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;

 /* Преобразование двоичного представления в целое число
 */
 cout << "\"1000101011\" as number: "
 << bitset<100>(string("1000101011")).to_ulong() << endl;
}
```

В зависимости от разрядности типов **short** и **long** примерный результат выглядит так:

```
267 as binary short: 0000000100001011
267 as binary long: 000000000000000000000000100001011
10.000.000 with 24 bits: 100110001001011010000000
"1000101011" as number: 555
```

В этом примере следующая конструкция преобразует число 267 в битовое поле, количество битов в котором определяется числом типа **unsigned short** (числовые пределы описаны на с. 72):

```
bitset<numeric_limits<unsigned short>::digits>(267)
```

Оператор вывода **bitset** выводит содержимое битового поля в виде последовательности нулей и единиц.

Аналогично, представленная ниже конструкция преобразует серию двоичных цифр в битовое поле, которое преобразуется в целое число вызовом **to\_ulong()**:

```
bitset<100>(string("1000101011"))
```

Обратите внимание: количество битов в битовом поле должно быть меньше `sizeof(unsigned long)`. Если значение битового поля не может быть представлено в виде `unsigned long`, генерируется исключение<sup>1</sup>.

## Строение класса `bitset`

Ниже перечислены операции, поддерживаемые классом `bitset`.

### Операции создания, копирования и уничтожения

Для битовых полей определено несколько специальных конструкторов. С другой стороны, для них не определены специальный копирующий конструктор, оператор присваивания и деструктор. Это означает, что присваивание и копирование битовых полей производится стандартными операциями поразрядного копирования.

`bitset<размер>::bitset ()`

- Конструктор по умолчанию
- Создает битовое поле, в котором все биты инициализированы нулями.
- Пример:

```
bitset<50> flags; // Флаги: 0000...000000
 // Всего 50 нулевых битов
```

`bitset<размер>::bitset (unsigned long value)`

- Создает битовое поле и инициализирует его битами целочисленного значения `value`.
- Если количество битов в `value` слишком мало, недостающие начальные биты инициализируются нулями.
- Пример:

```
bitset<50> flags(7); // Флаги: 0000...000111
```

```
explicit bitset<размер>::bitset (const string& str)
bitset<размер>::bitset (const string& str, string::size_type str_idx)
bitset<размер>::bitset (const string& str, string::size_type str_idx,
 string::size_type str_num)
```

- Все формы возвращают битовое поле, инициализированное строкой `str` или входящей в нее подстрокой.

<sup>1</sup> Отметьте необходимость явного преобразования исходного значения к типу `string`. Вероятно, это объясняется ошибкой в стандарте, поскольку в ранних версиях стандарта допускалось использование конструкций вида `bitsetd("1000101011")`. При преобразовании конструктора в шаблон для разных строковых типов этот механизм автоматического преобразования типа был случайно утрачен. В настоящее время рассматривается предложение по исправлению этой ошибки.

- Стока или подстрока может содержать только символы «0» и «1».
- Параметр `str_idx` определяет индекс первого символа `str`, используемого для инициализации.
- Если параметр `str_num` не задан, используются все символы от позиции `str_idx` до конца строки.
- Если количество символов в строке или подстроке меньше необходимого, начальные биты инициализируются нулями.
- Если количество символов в строке или подстроке больше необходимого, лишние символы игнорируются.
- Если выполняется условие `str_idx > str.size()`, генерируется исключение `out_of_range`.
- Если хотя бы один символ отличен от «0» и «1», генерируется исключение `invalid_argument`.
- Обратите внимание: конструктор объявлен в виде шаблонной функции класса (см. с. 28), из-за чего отсутствует неявное преобразование типа первого параметра из `const char*` в `string`<sup>1</sup>.
- Пример:

```
bitset<50> flags(string("1010101")); // Флаги: 0000...0001010101
bitset<50> flags(string("1111000")); // Флаги: 0000...0000000110
```

## Немодифицирующие операции

`size_t bitset<размер>::size () const`

Возвращает количество битов (то есть *размер*).

`size_t bitset<размер>::count () const`

Возвращает количество установленных битов (то есть битов со значением 1).

`bool bitset<размер>::any () const`

Проверяет наличие хотя бы одного установленного бита.

`bool bitset<размер>::none () const`

Проверяет отсутствие хотя бы одного установленного бита.

`bool bitset<размер>::test (size_t idx) const`

- Проверяет установку бита в позиции *idx*.

- Если выполняется условие `idx > =size()`, генерируется исключение `out_of_range`.

---

<sup>1</sup> Вероятно, это объясняется ошибкой в стандарте, поскольку в ранних версиях стандарта допускалось использование конструкций вида `bitset2 flags("1010101")`. При преобразовании конструктора в шаблон для разных строковых типов этот механизм автоматического преобразования типа был случайно утрачен. В настоящее время рассматривается предложение по исправлению этой ошибки.

```
bool bitset<размер>::operator== (const bitset<размер>& bits) const
```

Проверяет совпадение всех битов `*this` и `bits`.

```
bool bitset<размер>::operator!= (const bitset<размер>& bits) const
```

Проверяет наличие несовпадающих битов в `*this` и `bits`.

## Модифицирующие операции

```
bitset<размер>& bitset<размер>::set ()
```

- Устанавливает все биты.
- Возвращает модифицированное битовое поле.

```
bitset<размер>& bitset<размер>::set (size_t idx)
```

- Устанавливает бит в позиции `idx`.
- Возвращает модифицированное битовое поле.
- Если выполняется условие `idx > size()`, генерируется исключение `out_of_range`.

```
bitset<размер>& bitset<размер>::set (size_t idx, int value)
```

- Задает значение бита в позиции `idx` в соответствии с `value`.
- Возвращает модифицированное битовое поле.
- Значение `value` интерпретируется как логическая величина. Если аргумент `value` равен 0, бит сбрасывается, а при любом другом значении бит устанавливается.
- Если выполняется условие `idx > size()`, генерируется исключение `out_of_range`.

```
bitset<размер>& bitset<размер>::reset ()
```

- Сбрасывает все биты (то есть присваивает им 0).
- Возвращает модифицированное битовое поле.

```
bitset<размер>& bitset<размер>::reset (size_t idx)
```

- Сбрасывает бит в позиции `idx`.
- Возвращает модифицированное битовое поле.
- Если выполняется условие `idx > size()`, генерируется исключение `out_of_range`.

```
bitset<размер>& bitset<размер>::flip ()
```

- Переводит все биты в противоположное состояние (установленные биты сбрасываются, и наоборот).
- Возвращает модифицированное битовое поле.

```
bitset<размер>& bitset<размер>::flip (size_t idx)
```

- Переводит бит в позиции `idx` в противоположное состояние.
- Возвращает модифицированное битовое поле.

`bitset<размер>& bitset<размер>::operator^= (const bitset<размер>& bits)`

- Поразрядный оператор «исключающего ИЛИ».
- Переводит в противоположное состояние все биты, установленные в *bits*. Остальные биты остаются без изменений.
- Возвращает модифицированное битовое поле.

`bitset<размер>& bitset<размер>::operator|= (const bitset<размер>& bits)`

- Поразрядный оператор «ИЛИ».
- Устанавливает все биты, установленные в *bits*. Остальные биты остаются без изменений.
- Возвращает модифицированное битовое поле.

`bitset<размер>& bitset<размер>::operator&= (const bitset<размер>& bits)`

- Поразрядный оператор «И».
- Сбрасывает все биты, сброшенные в *bits*. Остальные биты остаются без изменений.
- Возвращает модифицированное битовое поле.

`bitset<размер>& bitset<размер>::operator<<= (size_t num)`

- Сдвигает все биты на *num* позиций влево.
- Возвращает модифицированное битовое поле.
- Последние *num* битов заполняются нулями.

`bitset<размер>& bitset<размер>::operator>>= (size_t num)`

- Сдвигает все биты на *num* позиций вправо.
- Возвращает модифицированное битовое поле.
- Первые *num* битов заполняются нулями.

## Работа с отдельными битами с применением оператора []

`bitset<размер>::reference bitset<размер>::operator[](size_t idx)`  
`bool bitset<размер>::operator[](size_t idx) const`

- Обе формы возвращают значение бита в позиции *idx*.
- Чтобы возвращаемое значение могло использоваться как модифицируемый объект (l-значение), первая форма для неконстантных битовых полей использует специальные временные объекты — *заместители* (*proxy*). Подробности приводятся далее.
- Перед вызовом необходимо убедиться в том, что *idx* представляет действительный индекс; в противном случае вызов приводит к непредсказуемым последствиям.

При вызове для неконстантных битовых полей оператор [] возвращает специальный временный объект типа `bitset<>::reference`. Этот объект используется

в качестве заместителя<sup>1</sup> для выполнения некоторых действий с битом, полученным при помощи оператора []. В частности, для `reference` определены пять перечисленных ниже операций.

`reference& operator= (bool)`

Задает значение бита в соответствии с переданной величиной.

`reference& operator= (const reference&)`

Задает значение бита по другой ссылке.

`reference& flip ()`

Переводит бит в противоположное состояние.

`operator bool () const`

Преобразует значение в логическую величину (автоматически).

`bool operator~ () const`

Возвращает дополнение (противоположное состояние) бита.

Примеры выполнения операций с возвращаемым значением оператора []:

```
bitset<50> flags;
...
flags[42] = true; // Установка бита 42
flags[13] = flags[42]; // Присваивание значения бита 42 биту 13
flags[42].flip(); // Перевод бита 42 в противоположное состояние
if (flags[13]) { // Если бит 13 установлен,
 flags[10] = ~flags[42]; // присвоить дополнение бита 42 биту 10
}
```

## Создание модифицированных битовых полей

`bitset<размер> bitset<размер>::operator~ () const`

Возвращает новое битовое поле, в котором все биты находятся в противоположном состоянии по отношению к `*this`.

`bitset<размер> bitset<размер>::operator<< (size_t num) const`

Возвращает новое битовое поле, в котором все биты сдвинуты влево на `num` позиций.

`bitset<размер> bitset<размер>::operator>> (size_t num) const`

Возвращает новое битовое поле, в котором все биты сдвинуты вправо на `num` позиций.

---

<sup>1</sup> Заместители позволяют выполнять операции с объектами, для которых в обычных условиях такая возможность отсутствует. В частности, они часто используются для защиты данных. В данном случае заместитель позволяет выполнять с возвращаемым значением некоторые операции битовых полей, хотя в принципе оно ведет себя как обычный объект типа `bool`.

```
bitset<размер> operator& (const bitset<размер>& bits1,
 const bitset<размер>& bits2)
```

Возвращает поразрядную конъюнкцию *bits1* и *bits2* — новое битовое поле, в котором установлены только биты, установленные в *bits1* и *bits2*.

```
bitset<размер> operator| (const bitset<размер>& bits1,
 const bitset<размер>& bits2)
```

Возвращает поразрядную дизъюнкцию *bits1* и *bits2* — новое битовое поле, в котором установлены биты, установленные в *bits1* или в *bits2*.

```
bitset<размер> operator^ (const bitset<размер>& bits1,
 const bitset<размер>& bits2)
```

Возвращает поразрядную исключающую дизъюнкцию *bits1* и *bits2* — новое битовое поле, в котором установлены биты, установленные в *bits1*, но не в *bits2*, или наоборот.

## Операции преобразования типа

```
unsigned long bitset<размер>::to_ulong () const
```

- Возвращает целое число, двоичное представление которого определяется битовым полем.
- Если целое число не может быть представлено типом `unsigned long`, генерируется исключение `overflow_error`.

```
string bitset<размери>::to_string () const
```

- Возвращает строку с двоичным представлением битового поля, записанным символами «0» и «1» (для сброшенных и установленных битов соответственно).
- Символы строки следуют в порядке убывания индекса битов.
- Функция оформлена как шаблон, параметризируемый только по типу возвращаемого значения. Согласно правилам языка, необходимо использовать запись вида:

```
bitset<50> b;
...
b.template to_string<char,char_traits<char>,allocator<char> >()
```

## Операции ввода-вывода

```
istream& operator>> (istream& strm, bitset<размер>& bits)
```

- Читает битовое поле *bits* как последовательность символов «0» и «1».
- Чтение продолжается до тех пор, пока не будет выполнено одно из следующих условий:
  - прочитано максимальное количество символов (*размер*);
  - при чтении *strm* обнаружен конец файла;
  - следующий символ отличен от «0» или «1».

- » Возвращает *strm*.
- » Если количество прочитанных битов меньше количества битов в битовом поле, начальные биты заполняются нулями.
- » Если оператору не удается прочитать ни одного символа, он устанавливает для *strm* условие `ios::failbit`, в результате чего может быть сгенерировано соответствующее исключение (см. с. 576).

`ostream& operator<< (ostream& strm, const bitset<размер>& bits)`

- » Выводит в поток данных битовое поле *bits* в виде символьного двоичного представления (то есть в виде последовательности символов «0» и «1»).
- » Выходные символы создаются функцией `to_string()` (см. ранее).
- » Возвращает *strm*.

Пример использования приведен на с. 445

# 11 Строки

Эта глава посвящена строковым типам стандартной библиотеки C++. В ней описан базовый шаблон `basic_string<>` и его специализации `string` и `wstring`.

При работе со строками у программистов часто возникают недоразумения. Как правило, это происходит из-за того, что термин «строка» может означать совершенно разные вещи — обычный символьный массив типа `char*` (с квалификатором `const` или без него), экземпляр класса `string`. Это также может быть обобщающим названием для объектов, которые содержат строковую информацию. В этой главе термин «строка» означает объект любого из строковых типов стандартной библиотеки C++ (`string` или `wstring`). «Традиционные» же строки типов `char*` и `const char*` будут называться *C-строками*.

Учтите, что тип строковых литералов (например, `"hello"`) был заменен на `const char*`. Тем не менее для обеспечения совместимости поддерживается неявное (хотя и нежелательное) преобразование к `char*`.

## Общие сведения

Строковые классы стандартной библиотеки C++ позволяют работать со строками как с обычными типами, не создающими проблем для пользователей. Это означает, что строки можно копировать, присваивать и сравнивать как базовые типы, не беспокоясь о возможной нехватке памяти или размерах внутреннего блока, предназначенного для хранения символов. Вы просто используете нужный оператор, например, `=` (присваивание), `==` (проверка на равенство) или `+` (конкатенация). Короче говоря, строковые типы стандартной библиотеки C++ спроектированы так, чтобы они работали как базовые типы данных, не вызывающие никаких дополнительных проблем (во всяком случае, теоретически). Современная обработка данных во многом ориентирована на работу с текстом, поэтому данный аспект особенно важен для программистов с опытом работы на C, Fortran и других языках, в которых обработка строк реализована весьма неудобно.

Два примера, представленных ниже, демонстрируют возможности и применение строковых классов. Они написаны только для учебных целей и не имеют практической ценности.

## Пример построения имени временного файла

В первом примере строится имя временного файла по аргументам командной строки. Например, рассмотрим следующую команду:

```
string1 prog.dat mydir hello. oops.tmp end.dat
```

Если запустить программу с этими аргументами, то результат будет выглядеть так:

```
prog.dat => prog.tmp
mydir => mydir.tmp
hello. => hello.tmp
oops.tmp => oops.xxx
end.dat => end.tmp
```

В большинстве случаев временный файл снабжается расширением .tmp, а при передаче имени временного файла в аргументе вместо него используется 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) {
 // Аргумент интерпретируется как имя файла
 filename = argv[i];

 // Поиск точки в имени файла
 string::size_type idx = filename.find('.');
 if (idx == string::npos) {
 // Имя файла не содержит точек
 tmpname = filename + '.' + suffix;
 }
 else {
 /* Разделение имени файла на базовое имя и расширение
 * - базовое имя содержит все символы перед точкой
 * - расширение содержит все символы после точки
 */
 basename = filename.substr(0, idx);
 extname = filename.substr(idx+1);
 if (extname.empty()) {
```

```

 // Содержит точку, но без расширения:
 // присоединить символы tmp
 tmpname = filename;
 tmpname += suffix;
}
else if (extname == suffix) {
 // Расширение tmp заменяется на xxx
 tmpname = filename;
 tmpname.replace (idx+1, extname.size(), "xxx");
}
else {
 // Замена любого расширения на tmp
 tmpname = filename;
 tmpname.replace (idx+1, string::npos, suffix);
}
}

// Вывод имен исходного и временного файлов
cout << filename << " => " << tmpname << endl;
}
}

```

Сначала показанная ниже директива включает заголовочный файл стандартных строковых классов C++:

```
#include <string>
```

Как обычно, эти классы объявляются в пространстве имен **std**.

Следующее объявление создает четыре строковые переменные:

```
string filename, basename, extname, tmpname;
```

Аргументы не передаются, поэтому инициализация выполняется конструктором по умолчанию класса **string**. Конструктор по умолчанию инициализирует переменные пустыми строками.

Следующее объявление создает константную строку **suffix** со стандартным расширением для временных файлов:

```
const string suffix("tmp");
```

Переменная инициализируется обычной С-строкой и получает значение **tmp**. С-строки могут комбинироваться с объектами класса **string** практически всегда, когда могут использоваться два объекта **string**. В частности, все вхождения переменной **suffix** в программе можно заменить С-строками "tmp".

При каждой итерации цикла **for** показанная ниже команда присваивает новое значение строковой переменной **filename**:

```
filename = argv[1];
```

В данном случае новое значение определяется обычной С-строкой, однако оно также может определяться другим объектом класса **string** или отдельным символом (**char**).

Следующая команда ищет в строке `filename` первое вхождение символа . (точка):

```
string::size_type idx = filename.find('.');
```

Функция `find()` входит в группу функций, предназначенных для поиска в строках. Другие функции этой группы позволяют выполнить поиск в обратном направлении, поиск подстрок, ограничить поиск определенной частью строки или найти вхождение одного из нескольких возможных символов. Все поисковые функции возвращают не итератор, а целочисленный индекс позиции первого совпадения. Стандартный интерфейс строк не соответствует интерфейсу шаблонов STL, хотя итераторы могут работать и со строками (см. с. 480). Возвращаемое значение всех функций относится к типу `string::size_type` – целочисленному беззнаковому типу, определяемому в строковом классе<sup>1</sup>. Как обычно, индекс первого символа равен 0, а индекс последнего символа – *numberOfCharacters*-1. Помните, что индекс *numberOfCharacters* не является допустимым индексом. В отличие от C-строк объекты класса `string` не завершаются специальным символом «\0».

Признак неудачи при поиске возвращается в виде специального значения `npos`, которое также определяется строковым классом. Таким образом, следующая строка проверяет, была ли найдена точка в имени файла:

```
f (idx == string::npos)
```

Тип и значение `npos` часто становятся причиной ошибок при работе со строками. Будьте внимательны и всегда используйте следующую конструкцию при проверке возвращаемого значения функции поиска (а не `int` или `unsigned int`):

```
string::size_type
```

В противном случае сравнение с `string::npos` может не сработать. За подробностями обращайтесь на с. 478.

Если поиск точки оказывается неудачным, значит, имя файла не содержит расширения. В этом случае имя временного файла строится из исходного имени файла, точки и заранее определенного расширения для временных файлов:

```
tmpname = filename + '.' + suffix;
```

Конкатенация двух строк производится при помощи обычного оператора `+`. В конкатенации также могут участвовать C-строки и одиночные символы.

При обнаружении точки используется секция `else`. В этом случае имя файла по индексу найденного символа разбивается на две части: базовое имя и расширение. Разбиение производится функцией `substr()`:

```
basename = filename.substr(0, idx);
extname = filename.substr(idx+1);
```

В первом аргументе функции `substr()` передается начальный индекс. Необязательный второй аргумент определяет количество символов (а не конечный индекс!) Если второй аргумент отсутствует, в возвращаемую подстроку включаются все оставшиеся символы строки.

---

<sup>1</sup> В частности, тип `size_type` для строк зависит от модели памяти, используемой строковым классом. Подробности приведены на с. 508.

Всюду, где в аргументах передаются индекс и длина, выполняются представленные ниже два правила.

- Аргумент, определяющий *индекс*, должен быть действительным для данной строки. Его значение должно быть меньше количества символов в строке (как обычно, индекс первого символа равен 0). Кроме того, для определения конца может использоваться индекс позиции за последним символом.

Как правило, при обращении по индексу, превышающему фактическое количество символов, генерируется исключение `out_of_range`.

- Аргумент, определяющий *количество символов*, может иметь произвольное значение. Если он больше количества оставшихся символов в строке, то используются все оставшиеся символы. В частности, конструкция `string::npos` всегда может обозначать «все оставшиеся символы».

Следовательно, при отсутствии точки в следующем выражении произойдет исключение:

```
filename.substr(filename.find('.'))
```

С другой стороны, показанное ниже выражение будет выполнено без исключений:

```
filename.substr(0, filename.find('.'))
```

Если точка не найдена, возвращается все имя файла.

Но даже при наличии точки расширение, возвращаемое при вызове `substr()`, может оказаться пустым, если за точкой нет ни одного символа. Данная ситуация проверяется условием:

```
if (extname.empty())
```

Если условие истинно, то имя временного файла состоит из обычного имени файла с заранее определенным расширением:

```
tmpname = filename;
tmpname += suffix;
```

Расширение присоединяется оператором `+=`.

Также нельзя исключать, что имя файла уже содержит расширение для временных файлов. Чтобы выявить эту ситуацию, мы сравниваем две строки оператором `==` следующим образом:

```
if (extname == suffix)
```

Если это условие истинно, то обычное расширение временных файлов заменяется расширением `xxx`:

```
tmpname = filename;
tmpname.replace (idx+1, extname.size(), "xxx");
```

Выражение `extname.size()` возвращает количество символов в строке `extname`. Вместо `size()` также можно воспользоваться эквивалентной функцией `length()`. Таким образом, обе функции `size()` и `length()` возвращают количество символов.

Помните, что функция `size()` не имеет отношения к объему памяти, выделяемой для хранения строки<sup>1</sup>.

После проверки всех специальных условий происходит основная обработка данных. Расширение файла заменяется расширением `.tmp`, обычно используемым для временных файлов:

```
tmpname = filename;
tmpname.replace (idx+1, string::npos, suffix);
```

Конструкция `string::npos` в данном случае обозначает «все остальные символы». Иначе говоря, все символы после точки заменяются строкой `suffix`. Замена работает и в том случае, если имя файла содержит точку без дальнейших символов — «пустое расширение» заменяется строкой `suffix`.

Команда вывода имен исходного и временного файлов показывает, что для отображения строк могут применяться стандартные потоковые операторы вывода:

```
cout << filename << " => " << tmpname << endl;
```

## Пример чтения слов и вывода символов в обратном порядке

Во втором примере отдельные слова читаются из стандартного входного потока данных и символы каждого слова выводятся в обратном порядке. Слова разделяются стандартным символом пропуска (пробелом, новой строкой, табуляцией), запятой, точкой с запятой или двоеточием.

```
// string/string2.cpp
#include <iostream>
#include <string>
using namespace std;

int main (int argc, char** argv)
{
 const string delims(" \t.,;:");
 string line;

 // Для каждой успешно прочитанной строки
 while (getline(cin, line)) {
 string::size_type begIdx, endIdx;

 // Поиск начала первого слова
 begIdx = line.find_first_not_of(delims);
 // Пока удается найти начало очередного слова...
 while (begIdx != string::npos) {
```

<sup>1</sup> Существование двух эквивалентных функций связано с фактическим слиянием двух архитектурных подходов. Функция `length()` возвращает длину строки и является аналогом функции `strlen()` для обычных С-строк, тогда как функция `size()` поддерживается всеми контейнерными классами и возвращает количество элементов в соответствии с архитектурными канонами STL.

```

// Поиск конца текущего слова
endIdx = line.find_first_of (delims, begIdx);
if (endIdx == string::npos) {
 // Конец слова совпадает с концом строки
 endIdx = line.length();
}

// Вывод символов в обратном порядке
for (int i=endIdx-1; i>=static_cast<int>(begIdx); --i) {
 cout << line[i];
}
cout << ' ';

// Поиск начала следующего слова
begIdx = line.find_first_not_of (delims, endIdx);
}
cout << endl;
}
}

```

В этой программе все символы, используемые для разделения слов, определяются в специальной строковой константе:

```
const string delims(" \t.:");
```

Символ новой строки тоже используется как разделитель. Тем не менее он не требует особой обработки, поскольку программа читает данные по строкам.

Внешний цикл работает до тех пор, пока в переменную `line` успешно читается очередная строка:

```
string line;

while (getline(cin, line)) {
...
}
```

Функция `getline()` предназначена для чтения из потока данных в строковую переменную. Она читает все символы до ближайшего разделителя строк, которым по умолчанию является символ новой строки. Разделитель извлекается из потока данных, но не присоединяется к прочитанным данным. Передавая собственный разделитель строк в необязательном третьем аргументе, можно перевести функцию `getline()` в режим чтения лексем, разделенных заданными символами.

Внутри внешнего цикла производится поиск и вывод отдельных слов. Первая команда ищет начало первого слова:

```
begIdx = line.find_first_not_of(delims);
```

Функция `find_first_not_of()` возвращает индекс первого символа, не входящего в переданный строковый аргумент. Иначе говоря, функция возвращает первый символ, не указанный в переменной `delims` как разделитель. Как и прочие функции поиска, при отсутствии совпадения функция `find_first_not_of()` возвращает `string::npos`.

Внутренний цикл выполняется до тех пор, пока в потоке данных обнаруживается начало очередного слова:

```
while (begIdx != string::npos) {
 ...
}
```

Первая команда внутреннего цикла ищет конец текущего слова:

```
endIdx = line.find_first_of (delims, begIdx);
```

Функция `find_first_of()` ищет первое вхождение одного из символов, составляющих первый аргумент. Необязательный второй аргумент определяет позицию, с которой начинается поиск. В нашем случае он начинается за началом слова.

Если символ не найден, конец слова совпадает с концом строки:

```
if (endIdx == string::npos) {
 endIdx = line.length();
```

Количество символов определяется при помощи функции `length()`, которая для строк эквивалентна функции `size()`.

В следующей команде все символы слова выводятся в обратном порядке:

```
for (int i=endIdx-1; i>=static_cast<int>(begIdx); --i) {
 cout << line[i];
}
```

Для обращения к отдельным символам строки используется оператор `[]`. Помните, что этот оператор *не проверяет* действительность индекса. Это означает, что вы должны сами заранее убедиться в правильности индекса (как это сделано в нашем примере). Более безопасный способ обращения к символам основан на применении функции `at()`. Лишние проверки замедляют работу программы, поэтому обычно при обращении к символам указанная проверка не предусмотрена.

С индексами строк связана и другая неприятная проблема. Если забыть о приведении типа `begIdx` к типу `int`, возможно зацикливание или аварийное завершение программы. Как и в первом примере, это объясняется тем, что `string::size_type` является беззнаковым целым типом. Без преобразования типа знаковое значение `i` автоматически преобразуется в беззнаковое значение из-за сравнения с другим беззнаковым значением. В этом случае выражение `i>=begIdx` всегда равно `true`, если текущее слово начинается с начала строки. Дело в том, что переменная `begIdx` в этом случае равна 0, а любое беззнаковое значение всегда больше либо равно нулю. Программа зацикливается и прерывается только в результате сбоя при нарушении защиты памяти.

По этой причине автор старается избегать применения конструкций `string::size_type` и `string::npos`. На с. 478 описано более безопасное (хотя и не идеальное) обходное решение.

Последняя команда внутреннего цикла переводит `begIdx` к началу следующего слова, если его удается найти:

```
begIdx = line.find_first_not_of (delims, endIdx);
```

В отличие от первого вызова функции `find_first_not_of()` поиск начинается от конца предыдущего слова. Если предыдущее слово завершалось в конце строки, то `endIdx` содержит индекс конца строки. В этом случае поиск начнется от конца строки и вернет `string::npos`.

Попробуем запустить эту «полезную и нужную» программу для следующих входных данных:

```
pots & pans
I saw a reed
```

Результат выглядит так:

```
stop & snap
I was a deer
```

## Описание строковых классов

### Строковые типы

Все строковые типы и функции определяются в заголовочном файле `<string>`:

```
#include <string>
```

Как обычно, все идентификаторы принадлежат пространству имен `std`.

### Шаблонный класс `basic_string`

В файле `<string>` определяется базовый шаблон для всех строковых типов `basic_string`:

```
namespace std {
 template<class charT,
 class traits = char_traits<charT>,
 class Allocator = allocator<charT>>
 class basic_string;
}
```

Шаблон параметризуется по типу символов, трактовкам символьного типа и модели памяти.

- Первый параметр определяет тип данных отдельного символа.
- Необязательный второй параметр определяет класс трактовок, описывающих основные операции с символами строкового класса. В частности, класс трактовок задает способ копирования и сравнения символов (за подробностями обращайтесь на с. 659). Если класс трактовок не указан, используется класс трактовок по умолчанию для указанного типа символов. Пример пользовательского класса трактовок, позволяющего обрабатывать строки без учета регистра символов, приведен на с. 485.
- Третий необязательный аргумент определяет модель распределения памяти, используемую строковым классом. Как обычно, по умолчанию задействуется модель `allocator` (за подробностями обращайтесь на с. 49 и к главе 15)<sup>1</sup>.

---

<sup>1</sup> В системах, не поддерживающих параметры шаблонов по умолчанию, третий аргумент обычно отсутствует.

## Типы **string** и **wstring**

Стандартная библиотека C++ содержит две специализированные версии класса `basic_string<>`.

- **string** — специализированная версия шаблона для символов типа `char`:

```
namespace std {
 typedef basic_string<char> string;
}
```

- **wstring** — специализированная версия шаблона для символов типа `wchar_t`:

```
namespace std {
 typedef basic_string<wchar_t> wstring;
}
```

Эта версия позволяет работать со строками, содержащими символы в много-байтовой кодировке (например, в кодировке Unicode или в азиатских кодировках — проблемы интернационализации рассматриваются в главе 14).

В данной главе эти два типа строк не различаются. Принципы использования и возникающие проблемы остаются одинаковыми, поскольку все строковые классы обладают одинаковым интерфейсом. Таким образом, «строка» в данном контексте означает любой строковый тип, включая `string` и `wstring`. В приводимых примерах обычно используется тип `string`, потому что программы, как правило, пишутся в расчете на европейскую и англо-американскую языковую среду.

## Операции со строками

В табл. 11.1 перечислены все операции, определенные для строк.

**Таблица 11.1.** Операции со строками

| Операция                                                 | Описание                                                                  |
|----------------------------------------------------------|---------------------------------------------------------------------------|
| Конструкторы                                             | Создание и копирование строк                                              |
| Деструктор                                               | Уничтожение строк                                                         |
| <code>=, assign()</code>                                 | Присваивание нового значения                                              |
| <code>swap()</code>                                      | Обмен содержимым двух строк                                               |
| <code>+=, append(), push_back()</code>                   | Присоединение символов                                                    |
| <code>insert()</code>                                    | Вставка символов                                                          |
| <code>erase()</code>                                     | Удаление символов                                                         |
| <code>clear()</code>                                     | Удаление всех символов (строка остается пустой)                           |
| <code>resize()</code>                                    | Изменение количества символов (удаление и присоединение символов в конце) |
| <code>replace()</code>                                   | Замена символов                                                           |
| <code>+</code>                                           | Конкатенация строк                                                        |
| <code>==, !=, &lt;, &lt;=, &gt;, &gt;=, compare()</code> | Сравнение строк                                                           |

продолжение ↗

**Таблица 11.1** (продолжение)

| Операция                         | Описание                                                                              |
|----------------------------------|---------------------------------------------------------------------------------------|
| <code>size(), length()</code>    | Получение количества символов                                                         |
| <code>max_size()</code>          | Получение максимально возможного количества символов                                  |
| <code>empty()</code>             | Проверка пустой строки                                                                |
| <code>capacity()</code>          | Получение количества символов, которые могут храниться в памяти без перераспределения |
| <code>reserve()</code>           | Резервирование памяти для заданного количества символов                               |
| <code>[], at()</code>            | Обращение к символу                                                                   |
| <code>&gt;&gt;, getline()</code> | Чтение строковых данных из потока данных                                              |
| <code>&lt;&lt;</code>            | Запись строковых данных в поток данных                                                |
| <code>copy()</code>              | Копирование или запись содержимого строки в С-строку                                  |
| <code>c_str()</code>             | Получение содержимого строки в виде С-строки                                          |
| <code>data()</code>              | Получение содержимого строки в виде символьного массива                               |
| <code>substr()</code>            | Получение подстроки                                                                   |
| Поисковые функции                | Поиск заданных подстрок и символов                                                    |
| <code>begin(), end()</code>      | Поддержка «нормальных» итераторов                                                     |
| <code>rbegin(), rend()</code>    | Поддержка обратных итераторов                                                         |
| <code>get_allocator()</code>     | Получение распределителя памяти                                                       |

## Аргументы операций со строками

Стандартная библиотека поддерживает множество операций, предназначенных для работы со строками. В частности, операции, изменяющие содержимое строки, обычно существуют в нескольких перегруженных версиях, в которых новое значение задается одним, двумя или тремя аргументами. Во всех операциях такого рода используется схема передачи аргументов, представленная в табл. 11.2.

**Таблица 11.2.** Схема передачи аргументов строковых операций

| Операция                                                                         | Описание                                                |
|----------------------------------------------------------------------------------|---------------------------------------------------------|
| <code>const string&amp; str</code>                                               | Вся строка str                                          |
| <code>const string&amp; str, size_type idx,</code><br><code>size_type num</code> | Не более первых num символов str, начиная с индекса idx |
| <code>const char* str</code>                                                     | Вся С-строка                                            |
| <code>const char* chars, size_type len</code>                                    | Первые len символов символьного массива chars           |
| <code>char c</code>                                                              | Символ c                                                |
| <code>size_type num, char c</code>                                               | Num экземпляров символа c                               |
| <code>iterator beg, iterator end</code>                                          | Все символы в интервале [beg,end)                       |

Обратите внимание: только в одноаргументной версии с передачей `char*` символ `\0` интерпретируется как специальный символ, завершающий строку. В остальных случаях `\0` не считается специальным символом.

```
std::string s1("nico"); // s1 инициализируется символами n,i,c,o
std::string s2("nico",5); // s2 инициализируется символами n,i,c,o,\0
std::string s3(5,'\\0'); // s3 инициализируется символами \0,\0,\0,\0,\0

s1.length() // Результат равен 4
s2.length() // Результат равен 5
s3.length() // Результат равен 5
```

Таким образом, в общем случае строка состоит из произвольных символов. Например, в нее можно загрузить содержимое двоичного файла.

В табл. 11.3 приведена краткая сводка аргументов, используемых различными операциями.

**Таблица 11.3. Операции со строковыми параметрами**

|                                      | Полная строка | Часть строки<br>( <code>char*</code> ) | С-строка<br>( <code>char*</code> ) | Массив<br><code>char</code> | Отдельный символ | пят<br>символов | Интервал<br>итераторов |
|--------------------------------------|---------------|----------------------------------------|------------------------------------|-----------------------------|------------------|-----------------|------------------------|
| Конструкторы                         | Да            | Да                                     | Да                                 | Да                          | —                | Да              | Да                     |
| =                                    | Да            | —                                      | Да                                 | —                           | Да               | —               | —                      |
| assign()                             | Да            | Да                                     | Да                                 | Да                          | —                | Да              | Да                     |
| +=                                   | Да            | —                                      | Да                                 | —                           | Да               | —               | —                      |
| append()                             | Да            | Да                                     | Да                                 | Да                          | —                | Да              | Да                     |
| push_back()                          | —             | —                                      | —                                  | —                           | Да               | —               | —                      |
| insert(),<br>индексная<br>версия     | Да            | Да                                     | Да                                 | Да                          | —                | Да              | —                      |
| insert(),<br>итераторная<br>версия   | —             | —                                      | —                                  | —                           | Да               | Да              | Да                     |
| replace (),<br>индексная<br>версия   | Да            | Да                                     | Да                                 | Да                          | Да               | Да              | —                      |
| replace (),<br>итераторная<br>версия | Да            | —                                      | Да                                 | Да                          | —                | Да              | Да                     |
| Поисковые<br>функции                 | Да            | —                                      | Да                                 | Да                          | Да               | —               | —                      |
| +                                    | Да            | —                                      | Да                                 | —                           | Да               | —               | —                      |
| ==, !=, <,<br><=, >, >=              | Да            | —                                      | Да                                 | —                           | —                | —               | —                      |
| compare()                            | Да            | Да                                     | Да                                 | Да                          | —                | —               | —                      |

## Отсутствующие операции

Строковые классы стандартной библиотеки C++ не предназначены для решения некоторых задач, возникающих при работе со строками. В частности, в них не предусмотрены следующие возможности:

- поддержка регулярных выражений;
- обработка текста (изменение регистра символов, сравнения без учета регистра символов).

Впрочем, основные функции обработки текста легко реализовать самостоятельно. Примеры приведены на с. 480.

## Конструкторы и деструкторы

В табл. 11.4 перечислены конструкторы и деструктор строк. Инициализация по интервалу, заданному при помощи итераторов, описана на с. 480.

**Таблица 11.4.** Конструкторы и деструктор строк

| Выражение                                  | Описание                                                                                                                                                      |
|--------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>string s</code>                      | Создает пустую строку <code>s</code>                                                                                                                          |
| <code>string s(str)</code>                 | Создает новую строку как копию существующей строки <code>str</code>                                                                                           |
| <code>string s(str, stridx)</code>         | Создает строку <code>s</code> , инициализированную символами строки <code>str</code> , начиная с индекса <code>stridx</code>                                  |
| <code>string s(str, stridx, strlen)</code> | Создает строку <code>s</code> , инициализированную не более чем <code>strlen</code> символами строки <code>str</code> , начиная с индекса <code>stridx</code> |
| <code>string s(cstr)</code>                | Создает строку <code>s</code> , инициализированную С-строкой <code>cstr</code>                                                                                |
| <code>string s(chars, chars_len)</code>    | Создает строку <code>s</code> , инициализированную <code>chars_len</code> символами символьного массива <code>chars</code>                                    |
| <code>string s(num, c)</code>              | Создает строку, состоящую из <code>num</code> экземпляров символа <code>c</code>                                                                              |
| <code>string s(beg, end)</code>            | Создает строку, инициализированную всеми символами интервала <code>[beg,end]</code>                                                                           |
| <code>s.string()</code>                    | Уничтожает все символы и освобождает память                                                                                                                   |

Строка не может инициализироваться одним символом. Вместо этого необходимо использовать адрес или явно задать количество вхождений:

```
std::string s('x'); // ОШИБКА
std::string s(1,'x'); // OK, создает строку из одного символа 'x'
```

Следовательно, автоматическое преобразование к типу `string` может производиться от типа `const char*`, но не от типа `char`.

## Строки и С-строки

В стандарте C++ тип строковых литералов `char*` был заменен типом `const char*`. Впрочем, для сохранения совместимости поддерживается неявное, хотя и неже-

лательное преобразование к типу `char*`. Но так как строковые литералы все же не относятся к типу `string`, между «новыми» объектами строкового класса и традиционными С-строками существует тесная связь: С-строки могут использоватьсь практически в любых операциях вместе со строками (сравнение, присоединение, вставка и т. д.). В частности, поддерживается автоматическое преобразование типа `const char*` в строку. С другой стороны, *не поддерживается* автоматическое преобразование строковых объектов в С-строки. Возможность такого преобразования была исключена по соображениям безопасности — чтобы предотвратить непреднамеренные преобразования типов, приводящие к странным последствиям (а тип `char*` часто ведет себя довольно странно) и неоднозначности (например, в выражении, объединяющем строку `string` с С-строкой, можно было бы выполнить преобразование как `string` в `char*`, так и наоборот). Вместо этого в классе `string` были определены специальные функции для создания или записи/копирования С-строк. В частности, функция `c_str()` переводит содержимое строки в формат С-строки (то есть преобразует его в символьный массив с последним символом `\0`). Функция `copy()` позволяет копировать и записывать строковые значения в существующие С-строки и символьные массивы.

Следует помнить, что в строках *не существует* специальной интерпретации символа \0, который в традиционных С-строках является признаком конца строки. Символ \0 может входить в строки наравне с любым другим символом.

Также учитите, что вместо параметра `char*` нельзя передавать `NULL`-указатель — это приводит к странным последствиям. Дело в том, что `NULL` относится к целочисленному типу и в перегруженных операциях интерпретируется как число 0 или символ со значением 0.

Преобразование содержимого строки в массив символов или С-строку осуществляется тремя функциями.

- `data()`. Возвращает содержимое строки в виде массива символов. Возвращаемое значение *не является* действительной С-строкой, поскольку к нему не присоединяется символ `\0`.
  - `c_str()`. Возвращает содержимое строки в формате С-строки (то есть с завершающим символом `\0`).
  - `copy()`. Копирует содержимое строки в символьный массив, передаваемый при вызове. Завершающий символ `\0` не присоединяется.

Функции `data()` и `c_str()` возвращают массив, принадлежащий строке, поэтомузывающая сторона не должна модифицировать или освобождать память. Пример:

Обычно в программе следует работать со строками, а их преобразование в С-строки или символьные массивы должно производиться непосредственно перед тем, как вам потребуется содержимое строки в виде типа `char*`. Помните, что возвращаемые значения функций `c_str()` и `data()` остаются действительными только до следующего вызова неконстантной функции для той же строки:

```
std::string s;
...
foo(s.c_str()); // Результат c_str() остается действительным
 // на время выполнения команды
const char* p;
p = s.c_str(); // p ссылается на содержимое s в формате С-строки
foo(p); // OK (значение p остается действительным)
s += "ext"; // Значение p становится недействительным
foo(p); // ОШИБКА: недействительное значение аргумента p
```

## Размер и емкость

Чтобы правильно и эффективно использовать строки, необходимо хорошо понимать, как связаны их размер и емкость. Для строк существуют целых три «размера».

- `size()` и `length()`. Эти функции возвращают количество символов в строке и являются эквивалентными<sup>1</sup>.

Функция `empty()` проверяет, равно ли количество символов нулю (то есть является ли строка пустой). По возможности используйте ее вместо функций `length()` и `size()`, поскольку эта функция может работать быстрее.

- `max_size()`. Эта функция возвращает максимальное количество символов, которые могут содержаться в строке. Обычно все символы строки хранятся в одном блоке памяти, что может привести к дополнительным ограничениям. В остальных случаях это значение обычно равно максимальному значению типа индекса, уменьшенному на 1. Единица вычитается по двум причинам: во-первых, максимальное значение представляет `pos`, и, во-вторых, реализация может дописать `\0` в конец внутреннего буфера, чтобы при преобразовании строки в формат С-строк (например, при вызове `c_str()`) можно было просто вернуть этот буфер. Если в результате операции должна быть создана строка, длина которой превышает `max_size()`, класс генерирует исключение `length_error`.

- `capacity()`. Эта функция возвращает емкость строки, то есть количество символов, которые можно сохранить в строке без перераспределения ее внутренней памяти.

---

<sup>1</sup> В данном случае эквивалентность функций объясняется тем, что `length()` возвращает длину строки по аналогии с функцией `strlen()` для обычных С-строк, а функция `size()` поддерживается всеми контейнерными классами и возвращает количество элементов в соответствии с архитектурными канонами STL.

Достаточная емкость строки важна по двум причинам:

- в результате перераспределения памяти становятся недействительными все ссылки, указатели и итераторы, ссылающиеся на символы строки;
- на перераспределение памяти тратится время.

Следовательно, если в программе используются указатели, ссылки и итераторы, относящиеся к символам строки, или если программа критична по скорости, следует учитывать ее емкость.

Функция `reserve()` помогает предотвратить перераспределение памяти. Она заранее резервирует нужную емкость и обеспечивает действительность ссылок (при условии, что размер строки не превышает зарезервированную емкость):

```
std::string s; // Создание пустой строки
s.reserve(80); // Резервирование памяти для 80 символов
```

В принципе концепция емкости для строк похожа на концепцию емкости векторов (см. с. 157), но с одним важным отличием: строки позволяют вызывать функцию `reserve()` для сокращения емкости. Вызов `reserve()` с аргументом, меньшим текущей емкости, фактически является запросом на сокращение емкости, не обязательным для выполнения. Если аргумент меньше текущего количества символов, то емкость сокращается до текущего размера строки. Следовательно, даже если вы хотите сократить емкость, вызов функции еще не гарантирует желаемого результата. По умолчанию аргумент функции `reserve()` для строк равен нулю. Следовательно, вызов `reserve()` без аргументов всегда означает запрос на сокращение емкости до текущего размера строки, но этот запрос не обязательен для выполнения:

```
s.reserve(); // Запрос на сокращение емкости до текущего размера
```

Запрос на сокращение емкости не обязательен для выполнения, поскольку способ достижения оптимального быстродействия определяется реализацией. В реализациях строковых классов могут использоваться разные критерии оптимизации по скорости и затратам памяти. Следовательно, некоторые реализации могут увеличивать емкость с большими приращениями и вообще не поддерживать ее сокращения.

Тем не менее в стандарте указано, что емкость строки может уменьшаться только вследствие вызова функции `reserve()`. Это гарантирует, что ссылки, указатели и итераторы останутся действительными даже при удалении и изменении символов (если они относятся к позициям, предшествующим позиции модификации).

## Обращение к символам

Строки предоставляют возможность чтения и записи хранящихся в них символов. Обращение к отдельному символу производится одним из двух способов: оператором индексирования `[]` или функцией `at()`. Обе операции возвращают символ в позиции с заданным индексом. Как обычно, первому символу

соответствует индекс 0, а последнему – индекс `length()-1`. Тем не менее существуют и различия.

- Оператор [] не проверяет правильность индекса, передаваемого в аргументе, а функция at() выполняет такую проверку. При вызове функции at() с недействительным индексом генерируется исключение `out_of_range`. При вызове оператора [] с недействительным индексом возможны непредсказуемые последствия — скажем, недопустимые обращения к памяти, которые становятся причиной неприятных побочных эффектов, или аварийное завершение программы (кстати, это относительно удачный вариант, по крайней мере, вы будете знать, что в программе что-то не так).
  - Для константной версии оператора [] позиция за последним символом считается действительной. В этом случае текущее количество символов является допустимым индексом. Оператор возвращает значение, сгенерированное конструктором по умолчанию для типа символов. Следовательно, для объектов типа `string` возвращается символ \0.

Во всех остальных случаях (для неконстантной версии оператора [] и функции at()) индекс, равный текущему количеству символов, считается недействительным. Его использование приводит к исключениям или непредсказуемым последствиям.

### Пример:

```
const std::string cs("nico"); // cs содержит символы n,i,c,o
std::string s("abcde"); // s содержит символы a,b,c,d,e

s[2] // Символ с
s.at(2) // Символ с

s[100] // ОШИБКА: непредсказуемые последствия
s.at(100) // Исключение out_of_range

s[s.length()] // ОШИБКА: непредсказуемые последствия
cs[cs.length()] // Символ \0
s.at(s.length()) // Исключение out_of_range
cs.at(cs.length()) // Исключение out_of_range
```

Чтобы программа могла изменять отдельные символы строки, неконстантные версии [] и `at()` возвращают ссылки на символы. Помните, что при перераспределении памяти ссылки становятся недействительными:

```
s = "new long value"; // При перераспределении памяти r и p
 // становятся недействительными
r = 'X'; // ОШИБКА: непредсказуемые последствия
*p = 'Y'; // ОШИБКА: непредсказуемые последствия
```

Для предотвращения ошибок времени выполнения необходимо зарезервировать достаточную емкость функцией `reserve()` перед инициализацией `r` и `p`.

Ссылки и указатели, ссылающиеся на символы строки, становятся недействительными при выполнении следующих операций:

- обмен значений функцией `swap()`;
- чтение нового значения оператором `>>` или функцией `getline()`;
- экспорт содержимого строки функцией `data()` или `c_str()`;
- вызов любой из неконстантных функций класса `at()`, `begin()`, `rbegin()`, `end()` или `rend()` без оператора `[]`;
- вызов оператора `[]` после любой из функций `at()`, `begin()`, `rbegin()`, `end()` или `rend()`.

Данные условия также относятся к итераторам (см. с. 480).

## Операции сравнения

Для строк определены стандартные операторы сравнения. В качестве operandов могут использоваться как строки, так и С-строки:

```
std::string s1,s2;
...
s1 == s2 // Возвращает true, если s1 и s2 содержат одинаковые символы
s1 < "hello" // Проверяет, что s1 меньше С-строки "hello"
```

При сравнении строк операторами `<`, `<=`, `>` и `>=` символы сравниваются в лексикографическом порядке в соответствии с их текущими трактовками. Например, все следующие условия равны `true`:

```
std::string("aaaa") < std::string("bbbb")
std::string("aaaa") < std::string("abba")
std::string("aaaa") < std::string("aaaaaaaa")
```

При помощи функций `compare()` можно сравнивать подстроки. Эти функции позволяют использовать более одного аргумента для определения строк, поэтому подстрока может задаваться индексом и длиной. Помните, что функции `compare()` возвращают не логический признак, а целочисленный код. Ноль является признаком совпадения строк; отрицательное значение указывает на то, что `*this` меньше заданной строки, а положительное значение — что `*this` больше. Пример:

```
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")
```

Если сравнение должно производиться по другому критерию, вы можете определить этот критерий и воспользоваться алгоритмами сравнения STL (пример приводится на с. 480) или же применить специальные трактовки символов, позволяющие сравнивать строки без учета регистра символов. Но так как строки со специальным классом трактовок относятся к другому типу данных, они не могут комбинироваться с объектами типа `string` (см. пример на с. 485).

В программах, ориентированных на международный рынок, строки иногда приходится сравнивать в соответствии с локальным контекстом. Для этой цели в классе `locale` определен удобный оператор `()` (см. с. 676). Он использует фасет контекстного сравнения, предназначенный для сравнения строк с учетом национальных стандартов. За подробностями обращайтесь на с. 697.

## Модификация строк

Модификация строк производится различными операторами и функциями классов.

### Присваивание

Оператор `=` присваивает строке новое значение, заданное в виде строки, C-строки или отдельного символа. Если новое значение описывается несколькими аргументами, для его присваивания можно воспользоваться функцией `assign()`. Пример:

```
const std::string aString("othello");
std::string s;

s = aString; // Присваивание строки "othello"
s = "two\nlines"; // Присваивание C-строки
s = ' '; // Присваивание отдельного символа

s.assign(aString); // Присваивание "othello" (эквивалент оператора =)
s.assign(aString.1.3); // Присваивание подстроки "the"
s.assign(aString.2,std::string::npos); // Присваивание подстроки "hello"

s.assign("two\nlines"); // Присваивание C-строки (эквивалент оператора =)
s.assign("nico".5); // Присваивание символьного массива: n.i.c.o.\0
s.assign(5,'x'); // Присваивание символьного массива: x.x.x.x.x
```

Также строке можно присвоить интервал символов, заданный двумя итераторами. За подробностями обращайтесь на с. 480.

### Обмен значений

В типе `string`, как и во многих нетривиальных типах, определена специализированная версия функции `swap()`, которая меняет местами содержимое двух строк (глобальная функция `swap()` представлена на с. 81). Специализация `swap()` для строк гарантирует постоянную сложность, поэтому ее рекомендуется использовать для обмена строковых значений и присваивания, если присвоенная строка становится ненужной после выполнения операции.

## Очистка строк

Существует несколько способов удаления всех символов строки:

```
std::string s;

s = ""; // Присваивание пустой строки
s.clear(); // Очистка строки
s.erase(); // Удаление всех символов
```

## Вставка и удаление символов

В классе `string` определены многочисленные функции вставки, удаления и замены символов строки. Присоединение символов в конец строки выполняется оператором `+=`, функциями `append()` и `push_back()`. Пример:

```
const std::string aString("othello");
std::string s;

s += aString; // Присоединение строки "othello"
s += "two\nlines"; // Присоединение С-строки
s += '\n'; // Присоединение отдельного символа

s.append(aString); // Присоединение строки "othello" (эквивалент +=)
s.append(aString,1,3); // Присоединение подстроки "the"
s.append(aString,2,std::string::npos); // Присоединение подстроки "hello"

s.append("two\nlines"); // Присоединение С-строки (эквивалент +=)
s.append("nico",5); // Присоединение символьного массива: n,i,c,o,\0
s.append(5,'x'); // Присоединение пяти символов: x,x,x,x,x

s.push_back('\n'); // Присоединение одного символа (эквивалент +=)
```

Оператор `+=` присоединяет строковое значение, определяемое одним аргументом. Функция `append()` позволяет определить присоединяемое значение несколькими аргументами. Одна из перегруженных версий `append()` присоединяет к строке интервал символов, заданный двумя итераторами (см. с. 480). Поддержка функции `push_back()` была добавлена для конечных итераторов вставки, чтобы алгоритмы STL могли использоваться для присоединения символов к строкам (конечные итераторы вставки рассматриваются на с. 275, а пример их применения со строками приведен на с. 480).

Функция `insert()`, предназначенная для вставки символов, также существует в нескольких версиях. При вызове функции передается индекс символа, за которым вставляются новые символы:

```
const std::string aString("age");
std::string s("p");

s.insert(1,aString); // s:page
s.insert(1,"ersifl"); // s:persiflage
```

Не существует версии `insert()`, которая бы получала индекс и одиночный символ. Следовательно, при вызове необходимо передавать строку или дополнительное число:

```
s.insert(0, ' '); // ОШИБКА
s.insert(0, " "); // OK
```

Также можно попытаться использовать следующий вызов, но это означает потенциальную неоднозначность:

```
s.insert(0.1, ' '); // ОШИБКА: неоднозначность
```

Причина неоднозначности — функция `insert()` перегружена для двух сигнатур:

```
insert (size_type idx, size_type num, charT c); // Позиция определяется
 // индексом
insert (iterator pos, size_type num, charT c); // Позиция определяется
 // итератором
```

Для типа `string` тип `size_type` обычно определяется как `unsigned`, а тип `iterator` — как `char*`. В этом случае для первого аргумента 0 существуют два равноправных преобразования. Следовательно, для получения желаемого результата приходится использовать запись:

```
s.insert((std::string::size_type)0, 1, ' '); // OK
```

Вторая интерпретация этой неоднозначности является собой пример вставки символов с применением итераторов. Если вы хотите задать позицию вставки при помощи итератора, это можно сделать тремя разными способами: путем вставки отдельного символа, путем вставки заданного количества экземпляров одного символа и путем вставки интервала, заданного двумя итераторами (см. с. 480).

По аналогии с рассмотренными функциями функции `erase()` (удаление символов) и `replace()` (замена) тоже существуют в нескольких версиях. Пример:

```
std::string s = "i18n"; // s:i18n
s.replace(1, 2, "nternaliozatio"); // s:internationalization
s.erase(13); // s:international
s.erase(7, 5); // s:internal
s.replace(0, 2, "ex"); // s:external
```

Функция `resize()` изменяет количество символов в строке. Если новый размер, переданный в аргументе функции, меньше текущего количества символов, то лишние символы удаляются с конца строки. Если новый размер больше текущего количества символов, то новые символы добавляются в начало. Новый символ, добавляемый при увеличении размеров строки, передается в необязательном аргументе. Если он отсутствует, используется конструктор по умолчанию для типа символов (то есть `\0` для типа `char`).

## Подстроки и конкатенация

Функция `substr()` выделяет заданную подстроку в произвольной строке. Примеры:

```
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 ")
```

Конкатенация двух строк, С-строк или одной из них с одиночным символом может осуществляться оператором +. Например, рассмотрим такой фрагмент:

```
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;
```

Этот фрагмент выводит следующий результат:

```
i18n means: internationalization
```

## Операторы ввода-вывода

Для строк определены традиционные операторы ввода-вывода:

- оператор >> читает строку из входного потока данных;
- оператор << выводит строку в выходной поток данных.

Эти операторы работают со строковыми объектами практически так же, как с обычными С-строками. В частности, оператор >> выполняет следующие действия.

1. Все начальные пропуски игнорируются, если флаг skipws не установлен (см. с. 600).
2. Оператор читает все последующие символы, пока не будет выполнено одно из следующих условий:
  - следующий символ является пропуском;
  - поток данных переходит в состояние, не позволяющее продолжать чтение (например, при достижении конца файла);
  - текущая ширина поля в потоке данных (см. с. 593) положительна, и из потока были прочитаны width() символов;
  - прочитаны max\_size() символов.
3. Ширина поля в потоке данных устанавливается равной нулю.

Отсюда следует, что в общем случае оператор ввода читает следующее слово, игнорируя начальные пропуски. Пропуском считается любой символ, для которого функция `isspace(c, stream.getloc())` возвращает true (функция `isspace()` рассматривается на с. 689).

Оператор вывода также учитывает ширину поля в потоке данных (это значение возвращает функция `width()`). Иначе говоря, если `width() > 0`, то оператор << выводит по меньшей мере `width()` символов.

Потоковые классы также определяют в пространстве имен `std` специальную функцию для построчного чтения данных: `std::getline()`. Функция читает все сим-

волы (вместе с начальными пропусками) до тех пор, пока не обнаружит разделитель строк или конец файла. Разделитель строк извлекается из потока данных, но не присоединяется к прочитанным данным. По умолчанию разделителем строк является символ новой строки, однако вы можете передать собственный разделитель в необязательном аргументе<sup>1</sup>. Таким образом, оператор << может использоваться для чтения лексем, разделенных произвольными символами:

```
std::string s;

while (getline(std::cin, s)) { // Для каждой строки, прочитанной из cin
 ...
}

while (getline(std::cin, s, ',')) { // Для каждой строки, прочитанной из cin
 ...
}
```

Учтите, что в режиме чтения лексем символ новой строки не является специальным символом. То есть лексемы могут содержать внутренние символы новой строки.

## Поиск

Для строк определены многочисленные функции поиска отдельных символов или подстрок. В частности, поддерживаются следующие возможности:

- поиск отдельных символов, последовательностей символов (подстрок) или вхождений одного символа из нескольких возможных;
- поиск в прямом и обратном направлениях;
- поиск, начиная с произвольной позиции (в начале или внутри строки).

Кроме того, итераторы позволяют передавать строки любым поисковым алгоритмам STL.

В именах всех поисковых функций присутствует слово «find». Эти функции ищут позицию символа со значением *value*, передаваемым в качестве аргумента. Конкретные особенности поиска зависят от точного названия поисковой функции. В табл. 11.5 перечислены все поисковые функции для строк.

**Таблица 11.5. Поисковые функции для строк**

| Функция         | Описание                                                               |
|-----------------|------------------------------------------------------------------------|
| find()          | Поиск первого вхождения <i>value</i>                                   |
| rfind()         | Поиск последнего вхождения <i>value</i> (поиск в обратном направлении) |
| find_first_of() | Поиск первого символа, входящего в <i>value</i>                        |

<sup>1</sup> Функцию `getline()` не обязательно снабжать префиксом `std::`, поскольку при вызове функции по «правилу Кенига» поиск автоматически начнется с того пространства имен, в котором определен класс аргумента.

| Функция                          | Описание                                                    |
|----------------------------------|-------------------------------------------------------------|
| <code>find_last_of()</code>      | Поиск последнего символа, входящего в <code>value</code>    |
| <code>find_first_not_of()</code> | Поиск первого символа, не входящего в <code>value</code>    |
| <code>find_last_not_of()</code>  | Поиск последнего символа, не входящего в <code>value</code> |

Все поисковые функции возвращают индекс первого символа последовательности, удовлетворяющей заданному условию. Если поиск завершается неудачей, все функции возвращают `npos`. Поисковые функции используют следующую схему передачи аргументов:

- в первом аргументе всегда передается искомое значение;
- второй аргумент определяет индекс, с которого должен начинаться поиск;
- необязательный третий аргумент определяет количество символов в искомом значении.

К сожалению, эта схема передачи аргументов отличается от схем, принятых в других строковых функциях, когда в первом аргументе передается начальный индекс, а значение и длина определяются соседними аргументами. Каждая строковая функция перегружается для перечисленных ниже аргументов.

`const string& value`

Поиск символов строки `value`.

`const string& value, size_type idx`

Поиск символов строки `value`, начиная с позиции `idx` в строке `*this`.

`const char* value`

Поиск символов С-строки `value`.

`const char* value, size_type idx`

Поиск символов С-строки `value`, начиная с позиции `idx` в строке `*this`.

`const char* value, size_type idx, size_type value_len`

Поиск `value_len` символов символьного массива `value`, начиная с позиции `idx` в строке `*this`. В этом случае символ `\0` не имеет специальной интерпретации внутри `value`.

`const char value`

Поиск символа `value`.

`const char value, size_type idx`

Поиск символа `value`, начиная с позиции `idx` в строке `*this`.

Пример:

```
std::string s("Hi Bill, I'm ill. so please pay the bill");
s.find("ill") // Возвращает 4 (первая подстрока "ill")
```

```
s.find("il",10) // Возвращает 13 (первая подстрока "il" после s[10])
s.rfind("il") // Возвращает 37 (последняя подстрока "il")
s.find_first_of("il") // Возвращает 1 (первый из символов i или l)
s.find_last_of("il") // Возвращает 39 (последний из символов i или l)
s.find_first_not_of("il") // Возвращает 0 (первый символ,
 // отличный от i и l)
s.find_last_not_of("il") // Возвращаетnpos
```

При поиске символов и подстрок также используются алгоритмы STL, позволяющие задавать пользовательские критерии сортировки (пример приведен на с. 478). Обратите внимание на отличия в именах поисковых алгоритмов STL и поисковых функций строк. За дополнительной информацией обращайтесь на с. 322.

## Значение `npos`

Если поиск не дает результатов, поисковые функции возвращают `string::npos`. Рассмотрим следующий пример:

```
std::string s;
std::string::size_type idx; // Будьте внимательны: другие типы
 // не подходят!
...
idx = s.find("substring");
if (idx == std::string::npos) {
 ...
}
```

Условие команды `if` равно `true` в том и только в том случае, если строка `s` не содержит подстроку "substring".

Будьте очень внимательны при использовании строкового значения `npos` и его типа. При проверке возвращаемых значений всегда имейте в виду тип возвращаемого значения `string::size_type`, но *не* `int` или `unsigned`; в противном случае сравнение возвращаемого значения с `string::npos` может не сработать.

Такое поведение объясняется тем, что при проектировании библиотеки значение `npos` было определено как число `-1`:

```
namespace std {
 template<class charT,
 class traits = char_traits<charT>,
 class 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 распределителем памяти строки) должен быть беззнаковым целым. Распределитель по умолчанию `allocator` использует в качестве `size_type` тип `size_t` (см. с. 706). Поскольку `-1` преобразуется к беззнаковому целому типу, `npos` является максимальным беззнаковым значением этого типа. Тем не менее точное значение зависит от конкретного определения типа `size_type`. К сожалению, на практике максимальные значения оказываются различными. Более того, `(unsigned long)-1` отличается от `(unsigned short)-1` (если различаются размеры этих типов). Следовательно, показанное ниже условие может оказаться равным `false`, если переменная `idx` равна `-1`, а `idx` и `string::npos` относятся к разным типам:

```
idx == std::string::npos
```

Пример:

```
std::string s;
...

int idx = s.find("not found"); // Предположим, возвращается npos
if (idx == std::string::npos) { // ОШИБКА: сравнение может не работать
 ...
}
```

Один из способов предотвращения этой ошибки основан на прямой проверке результата поиска:

```
if (s.find("hi") == std::string::npos) {
 ...
}
```

Однако довольно часто в программе нужен индекс позиции, в которой было найдено совпадение. Другое простое решение заключается в определении собственного знакового значения для `npos`:

```
const int NPOS = -1;
```

Сравнение принимает несколько иной (и даже более компактный) вид:

```
if (idx == NPOS) { // Работает почти всегда
 ...
}
```

К сожалению, это решение не идеально — проверка завершится неудачей, если `idx` относится к типу `unsigned short` или индекс превышает максимальное значение `int` (именно из-за этих проблем такое решение не было стандартизировано). Однако поскольку на практике обе ситуации встречаются очень редко, обычно это решение работает. Но если вы хотите, чтобы программа была действительно переносимой, всегда используйте тип `string::size_type` для всех индексов строковых типов. В идеальном решении вам придется определить перегруженные функции в зависимости от конкретного типа `string::size_type`. Вероятно, в будущем в стандарте появится более удачное решение этой проблемы.

## Поддержка итераторов для строк

Строка представляет собой упорядоченную последовательность символов. Соответственно в стандартную библиотеку C++ был включен интерфейс, позволяющий использовать строки как контейнеры STL<sup>1</sup>.

В частности, вызовом функций строковых классов можно получить итераторы для перебора символов строки. Если вы еще не знакомы с итераторами, считайте, что это своего рода псевдоуказатели, ссылающиеся на отдельные символы строки (по аналогии с указателями, ссылающимися на отдельные символы С-строк). При помощи итераторов можно перебрать все символы строки. Для этого достаточно воспользоваться алгоритмами, входящими в стандартную библиотеку C++ или определяемыми пользователем. Например, вы можете отсортировать символы в строке, переставить их в обратном порядке или найти символ с максимальным значением.

Строковые итераторы относятся к категории итераторов произвольного доступа. Это означает, что строковые итераторы поддерживают произвольный доступ к символам и могут использоваться со всеми алгоритмами (см. с. 105 и 257). Как обычно, типы строковых итераторов (`iterator`, `const_iterator` и т. д.) определяются самим строковым классом. Конкретный тип зависит от реализации, но обычно строковые итераторы определяются в виде обычных указателей. На с. 264 описаны проблемы, возникающие из-за различий между итераторами, реализованными в виде указателей, и итераторами, реализованными в виде классов.

Итераторы становятся недействительными при перераспределении памяти и при некоторых изменениях в данных, на которые они ссылаются. За подробностями обращайтесь на с. 471.

### Строковые функции для работы с итераторами

В табл. 11.6 перечислены все функции строковых классов, предназначенные для работы с итераторами. Как обычно, итераторы `beg` и `end` определяют полуоткрытый интервал, который включает `beg`, но не включает `end`; такие интервалы часто обозначаются `[beg,end)` (см. с. 96).

Чтобы обеспечить поддержку конечных итераторов вставки для строк, была определена функция `push_back()`. Конечные итераторы вставки рассматриваются на с. 277, а пример их использования со строками можно найти на с. 484.

**Таблица 11.6.** Строковые функции для работы с итераторами

| Выражение               | Описание                                                                                                                     |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <code>s.begin()</code>  | Возвращает итератор произвольного доступа для первого символа                                                                |
| <code>s.end()</code>    | Возвращает итератор произвольного доступа для позиции за последним символом                                                  |
| <code>s.rbegin()</code> | Возвращает обратный итератор для первого символа при переборе в обратном направлении (то есть для последнего символа строки) |

<sup>1</sup> Общие сведения об STL приводятся в главе 5.

| Выражение                           | Описание                                                                                                                                              |
|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| s.rend()                            | Возвращает обратный итератор для позиции за последним символом при переборе в обратном направлении (то есть для позиции перед первым символом строки) |
| string s(beg, end)                  | Создает строку, инициализированную всеми символами интервала [beg,end)                                                                                |
| s.append(beg, end)                  | Присоединяет к строке все символы интервала [beg,end)                                                                                                 |
| s.assign(beg, end)                  | Присваивает строке содержимое интервала [beg,end)                                                                                                     |
| s.insert(pos, c)                    | Вставляет символ с в позиции итератора pos и возвращает итератор для позиции нового символа                                                           |
| s.insert(pos,num,c)                 | Вставляет num экземпляров символа с в позиции итератора pos и возвращает итератор для позиции первого из вставленных символов                         |
| s.insert(pos, beg, end)             | Вставляет все символы интервала [beg,end) в позиции итератора pos                                                                                     |
| s.erase(pos)                        | Удаляет символ, на который ссылается итератор pos, и возвращает позицию следующего символа                                                            |
| s.erase(beg, end)                   | Удаляет все символы интервала [beg,end) и возвращает позицию следующего символа                                                                       |
| s.replace(beg, end, str)            | Заменяет все символы в интервале [beg,end) символами строки str                                                                                       |
| s.replace(beg,end,cstr)             | Заменяет все символы в интервале [beg,end) символами С-строки cstr                                                                                    |
| s.replace(beg, end, cstr, len)      | Заменяет все символы в интервале [beg,end) len символами символьного массива cstr                                                                     |
| s.replace(beg,end,num,c)            | Заменяет все символы в интервале [beg,end) num экземплярами символа с                                                                                 |
| s.replace(beg, end, newBeg, newEnd) | Заменяет все символы в интервале [beg,end) всеми символами интервала [newBeg,newEnd)                                                                  |

## Пример использования итераторов со строками

При помощи строковых итераторов можно преобразовать все символы строки к верхнему или нижнему регистру всего одной командой. Пример:

```
// string/iter1.cpp
#include <string>
#include <iostream>
#include <algorithm>
#include <cctype>
using namespace std;

int main()
{
 // Создание строки
 string s("The zip code of Hondelage in Germany is 38108");
 cout << "original: " << s << endl;
```

```

// Преобразование всех символов к нижнему регистру
transform (s.begin(), s.end(), // Источник
 s.begin(), // Приемник
 tolower); // Операция
cout << "lowered: " << s << endl;

// Преобразование всех символов к верхнему регистру
transform (s.begin(), s.end(), // Источник
 s.begin(), // Приемник
 toupper); // Операция
cout << "uppered: " << s << endl;
}

```

Результат выполнения программы выглядит так:

```

original: The zip code of Hondelage in Germany is 38108
lowered: the zip code of hondelage in germany is 38108
uppered: THE ZIP CODE OF HDNDELAGE IN GERMANY IS 38108

```

Обратите внимание на устаревшие функции `tolower()` и `toupper()` языка C, использующие универсальный локальный контекст. При работе с другими локальными контекстами следует задействовать новые формы функций `tolower()` и `toupper()`. Подробности приводятся на с. 689.

Следующий пример показывает, как при помощи STL задать собственные критерии поиска и сортировки. В этом примере производится поиск и сравнение строк без учета регистра символов.

```

// string/iter2.cpp
#include <string>
#include <iostream>
#include <algorithm>
using namespace std;

bool nocase_compare (char c1, char c2)
{
 return toupper(c1) == toupper(c2);
}

int main()
{
 string s1("This is a string");
 string s2("STRING");

 // Сравнение строк без учета регистра символов
 if (s1.size() == s2.size() && // Проверить совпадение размеров
 equal (s1.begin(), s1.end(), // Первая строка
 s2.begin(), // Вторая строка
 nocase_compare)) { // Критерий сравнения
 cout << "the strings are equal" << endl;
 }
}

```

```

else {
 cout << "the strings are not equal" << endl;
}

// Поиск без учета регистра символов
string::iterator pos;
pos = search (s1.begin(), s1.end(), // Стока, в которой ведется поиск
 s2.begin(), s2.end(), // Искомая подстрока
 nocase_compare); // Критерий сравнения
if (pos == s1.end()) {
 cout << "s2 is not a substring of s1" << endl;
}
else {
 cout << '\"' << s2 << "\"" is a substring of \""
 << s1 << "\"" (at index " << pos - s1.begin() << ")"
 << endl;
}
}

```

Перед вызовом `equal()` нужно проследить за тем, чтобы количество элементов во втором интервале было по крайней мере не меньше, чем в первом. Этим объясняется необходимость сравнения размеров; без этой проверки возможны непредсказуемые последствия.

В последней команде вывода индекс символа вычисляется как разность двух итераторов:

```
pos - s1.begin()
```

Такой способ вычисления работает, потому что строковые итераторы являются итераторами произвольного доступа. Преобразование индекса в итератор производится аналогично — достаточно просто прибавить значение индекса.

В приведенном примере определяется вспомогательная функция `nocase_compare()`, сравнивающая строки без учета регистра символов. Вместо нее также можно было бы воспользоваться комбинацией функциональных адаптеров и заменить вызов `nocase_compare` следующим выражением:

```
compose_f_gx_hy(equal_to<int>(),
 ptr_fun(toupper),
 ptr_fun(toupper))
```

За дополнительной информацией обращайтесь на с. 310 и 318.

При хранении строк в множествах или отображениях может потребоваться специальный критерий сортировки строк без учета регистра символов. На с. 218 приведен пример определения такого критерия.

В следующей программе приводятся другие примеры строковых операций с применением итераторов:

```
// string/iter3.cpp
#include <string>
#include <iostream>
```

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

int main()
{
 // Создание константной строки
 const string hello("Hello, how are you?");

 // Инициализация строки с всеми символами строки hello
 string s(hello.begin(), hello.end());

 // Перебор всех символов
 string::iterator pos;
 for (pos = s.begin(); pos != s.end(); ++pos) {
 cout << *pos;
 }
 cout << endl;

 // Перестановка символов строки в обратном порядке
 reverse (s.begin(), s.end());
 cout << "reverse: " << s << endl;

 // Сортировка всех символов строки
 sort (s.begin(), s.end());
 cout << "ordered: " << s << endl;

 /* Удаление смежных дубликатов
 * - unique() изменяет очередность символов и возвращает новый конец
 * - erase() удаляет лишние символы
 */
 s.erase (unique(s.begin(),
 s.end()),
 s.end());
 cout << "no duplicates: " << s << endl;
}
```

Результат выполнения программы выглядит так:

```
Hello, how are you?
reverse: ?uoy era woh ,olleH
ordered: .?Haeeh11oooruwy
no duplicates: .?Haehloruwy
```

В следующем примере содержимое строки читается из стандартного входного потока данных с использованием конечных итераторов вставки:

```
// string/unique.cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <locale>
```

```
using namespace std;

class bothWhiteSpaces {
private:
 const locale& loc; // Локальный контекст
public:
 /* Конструктор
 * - сохранение объекта локального контекста
 */
 bothWhiteSpaces (const locale& l) : loc(l) {
 }
 /* Функция проверяет, являются ли оба символа пропусками
 */
 bool operator() (char elem1, char elem2) {
 return isspace(elem1,loc) && isspace(elem2,loc);
 }
};

int main()
{
 string contents;

 // Не игнорировать начальные пропуски
 cin.unsetf (ios::skipws);

 // Чтение всех символов со сжатием пропусков
 unique_copy(istream_iterator<char>(cin), // Начало источника
 istream_iterator<char>(), // Конец источника
 back_inserter(contents), // Приемник
 bothWhiteSpaces(cin.getloc())); // Критерий удаления

 // Обработка содержимого
 // - в данном случае - зались в стандартный вывод
 cout << contents;
}
```

Алгоритм `unique_copy()` (см. с. 377) читает все символы из входного потока данных `cin` и вставляет их в строку `contents`. Объект функции `bothWhiteSpaces` проверяет, являются ли два последовательных символа пропусками. Для этого он инициализируется локальным контекстом `cin` и вызывает функцию `isspace()`, которая проверяет, относится ли символ к категории пропусков (функция `isspace()` рассматривается на с. 689). Алгоритм `unique_copy()` использует критерий `bothWhiteSpaces` для удаления смежных пропусков. Похожий пример приведен при описании алгоритма `unique_copy()` на с. 377.

## Интернационализация

Как упоминалось во вводном описании строковых классов (см. с. 462), шаблон строкового класса `basic_string<>` параметризуется по типу символов, трактовкам типа символов и модели памяти. Тип `string` представляет собой специализиро-

ванную версию шаблона для символов типа `char`, а тип `wstring` предназначен для символов типа `wchar_t`.

Трактовки типа символов представляют собой информацию о том, как следует поступать в зависимости от представления типа символов. Необходимость в дополнительном классе объясняется тем, что интерфейс встроенных типов (таких, как `char` и `wchar_t`) изменять нельзя, но один символьный тип может трактоваться по-разному. Дополнительная информация о классах трактовок приводится на с. 659.

В следующем фрагменте определяется специальный класс трактовок для строк, благодаря которому операции со строками выполняются без учета регистра символов:

```
// string/icstring.hpp
#ifndef ICSTRING_HPP
#define ICSTRING_HPP

#include <string>
#include <iostream>
#include <cctype>

/* Замена функций стандартного класса char_traits<char>
 * для того, чтобы операции со строками
 * выполнялись без учета регистра символов
 */
struct ignorecase_traits : public std::char_traits<char> {
 // Проверка равенства c1 и c2
 static bool eq(const char& c1, const char& c2) {
 return std::toupper(c1)==std::toupper(c2);
 }
 // Проверка условия "c1 меньше c2"
 static bool lt(const char& c1, const char& c2) {
 return std::toupper(c1)<std::toupper(c2);
 }
 // Сравнение до n символов 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;
 }
 }
 return 0;
 }
 // Поиск с в 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;
}

// Определение специального типа для таких строк
typedef std::basic_string<char, ignorecase_traits> icstring;

/* Определение оператора вывода,
 * так как тип трактовок отличен от типа,
 * заданного для std::ostream
 */
inline
std::ostream& operator << (std::ostream& strm, const icstring& s)
{
 // Простое преобразование icstring в обычную строку
 return strm << std::string(s.data(), s.length());
}

#endif // ICSTRING_HPP

```

Определение оператора вывода необходимо из-за того, что стандарт описывает операторы ввода-вывода только для потоков данных, у которых тип символов соответствует типу трактовок. В данном случае мы используем другой тип трактовок, поэтому для него приходится определять собственный оператор вывода. Аналогичная проблема существует и для операторов ввода.

В следующей программе показано, как использовать специализированный вид 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("A11");
 if (idx != icstring::npos) {
 cout << "index of \"A11\" in \""
 << s1 << "\" : "
 << idx << endl;
 }
}

```

```
 else {
 cout << "\"All\" not found in \"" << s1 << endl;
 }
}
```

Результат выполнения программы выглядит так:

```
hallo == otto : false
hallo == hALLo : true
index of "All" in "hallo" : 1
```

За дополнительными сведениями об интернационализации обращайтесь к главе 14.

## Эффективность

Стандарт не указывает, *как должен* быть реализован строковый класс, — он лишь определяет интерфейс строкового класса. В зависимости от выбранного концептуального подхода и приоритетов реализации могут существенно различаться по скорости работы и расходованию памяти.

Если вы добиваетесь оптимизации по скорости, используйте строковый класс, основанный на концепции *подсчета ссылок*. Подсчет ссылок ускоряет копирование и присваивание, поскольку реализация копирует и присваивает не содержимое строк, а только ссылки (на с. 226 описан класс умного указателя, реализующий подсчет ссылок для произвольного типа). Хотя при подсчете ссылок передача по константной ссылке может вам не потребоваться, для обеспечения нужной гибкости и переносимости программ такую передачу нужно обеспечить.

## Строки и векторы

Строки имеют много общего с векторами, и это не удивительно — оба типа контейнеров обычно реализуются в виде динамических массивов. Строку можно рассматривать как специализированную разновидность вектора с символьными элементами. Более того, строки могут использоваться как контейнеры STL (см. с. 480). Впрочем, интерпретировать строку как специализированный вектор опасно, поскольку между ними существует немало принципиальных различий, главное из которых — предназначение контейнера.

- Векторы предназначены для работы с элементами контейнеров, а не с контейнером в целом. По этой причине реализации векторов оптимизируются для эффективного выполнения операций с отдельными элементами.
- Строки предназначены для работы с целым контейнером (строкой), поэтому для них оптимизируются операции присваивания и передачи всего контейнера.

Различия в целях обычно приводят к совершенно разным реализациям. Например, строки часто реализуются на базе подсчета ссылок, а для векторов такая реализация вообще нехарактерна. Впрочем, вектор может использоваться как обычная С-строка. За подробностями обращайтесь на с. 164.

## Строение строковых классов

В этом разделе обозначение *string* соответствует фактическому строковому классу — *string*, *wstring* или любой другой специализированной версии класса *basic\_string*<>. Обозначение *char* относится к фактическому типу символов, то есть *char* для *string* и *wchar\_t* для *wstring*. Смысл других типов и значений, выделенных курсивом, зависит от определений типа символов и класса трактовок (см. с. 664).

### Определения типов и статические значения

*string::traits\_type*

- Тип трактовок символов.
- Второй параметр шаблона класса *basic\_string*.
- Для типа *string* — эквивалент *char\_traits*<*char*>.

*string::value\_type*

- Тип символов.
- Эквивалент *traits\_type::char\_type*.
- Для типа *string* — эквивалент *char*.

*string::size\_type*

- Беззнаковый целый тип для значений размеров и индексов.
- Эквивалент *allocator\_type::size\_type*.
- Для типа *string* — эквивалент *size\_t*.

*string::difference\_type*

- Знаковый целый тип для значений разности.
- Эквивалент *allocator\_type::difference\_type*.
- Для типа *string* — эквивалент *ptrdiff\_t*.

*string::reference*

- Тип ссылки на символ.
- Эквивалент *allocator\_type::reference*.
- Для типа *string* — эквивалент *char&*.

*string::const\_reference*

- Тип константной ссылки на символ.
- Эквивалент *allocator\_type::const\_reference*.
- Для типа *string* — эквивалент *const char&*.

*string::pointer*

- Тип указателя на символ.

- Эквивалент `allocator_type::pointer`.
- Для типа `string` — эквивалент `char*`.

`string::const_pointer`

- Тип константного указателя на символ.
- Эквивалент `allocator_type::const_pointer`.
- Для типа `string` — эквивалент `const char*`.

`string::iterator`

- Тип итераторов.
- Конкретный тип зависит от реализации.
- Для типа `string` обычно используется `char*`.

`string::reverse_iterator`

- Тип константных итераторов.
- Конкретный тип зависит от реализации.
- Для типа `string` обычно используется `const char*`.

`string::reverse_iterator`

- Тип обратных итераторов.
- Эквивалент `reverse_iterator<iterator>`.

`string::reverse_iterator`

- Тип константных обратных итераторов.
- Эквивалент `reverse_iterator<const_iterator>`.

`static const size_type string::npos`

- Специальное значение, означающее «безрезультатный поиск» или «все оставшиеся символы».
- Беззнаковое целое значение, инициализируемое значением `-1`.
- Использование значения `npos` требует осторожности. За подробностями обращайтесь к с. 478.

## Операции создания, копирования и уничтожения строк

`string::string ()`

- Конструктор по умолчанию.
- Создает пустую строку.

`string::string (const string& str)`

- Копирующий конструктор.
- Создает новую строку как копию `str`.

```
string::string (const string& str, size_type str_idx)
string::string (const string& str, size_type str_idx, size_type str_num)
```

- Создает новую строку, инициализированную не более чем *str\_num* символами *str*, начиная с индекса *str\_idx*.
- Если аргумент *str\_num* не указан, используются все символы от позиции *str\_idx* до конца *str*.
- Если *str\_idx*>*str.size()*, генерируется исключение `out_of_range`.

```
string::string (const char* cstr)
```

- Создает новую строку, инициализируемую содержимым С-строки *cstr*.
- Стока инициализируется всеми символами *cstr*, за исключением символа \0.
- Аргумент *cstr* не должен быть NULL-указателем.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение `length_error`.

```
string::string (const char* cstr, size_type chars_len)
```

- Создает новую строку, инициализируемую *chars\_len* символами символьного массива *chars*.
- Массив *chars* должен содержать не менее *chars\_len* символов. Символы могут иметь любые значения, поэтому символ \0 не имеет особой интерпретации.
- Если аргумент *chars\_len* равен `string::npos`, генерируется исключение `length_error`.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение `length_error`.

```
string::string (size_type num, char c)
```

- Создает строку, инициализируемую *num* экземплярами символа *c*.
- Если аргумент *num* равен `string::npos`, генерируется исключение `length_error`.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение `length_error`.

```
string::string (InputIterator beg, InputIterator end)
```

- Создает строку, инициализируемую всеми символами интервала `[beg,end)`.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение `length_error`.

```
string::~string ()
```

- Деструктор.
- Уничтожает все символы и освобождает память.

Большинство конструкторов позволяют в дополнительном аргументе передать распределитель памяти (см. с. 508).

## Операции с размером и емкостью

### Операции с размером

```
size_type string::size() const
size_type string::length() const
```

- Обе функции возвращают текущее количество символов в строке.
- Применительно к строкам они являются эквивалентными.
- Для проверки пустых строк следует использовать функцию `empty()`, поскольку она может работать быстрее.

```
bool string::empty () const
```

- Проверяет, является ли строка пустой (то есть не содержит ни одного символа).
- Эквивалент (но может работать быстрее):

```
string::size()==0
```

```
size_type string::max_size() const
```

- Возвращает максимальное количество символов, которые могут содержаться в строке.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение `length_error`.

### Операции с емкостью

```
size_type string::capacity () const
```

Возвращает количество символов, которые могут храниться в строке без перераспределения памяти.

```
void string::reserve ()
void string::reserve (size_type num)
```

- Вторая форма резервирует внутреннюю память, по крайней мере для *num* символов.
- Если *num* меньше текущей емкости, вызов интерпретируется как запрос на сокращение емкости, не обязательный к выполнению.
- Если *num* меньше текущего количества символов, вызов интерпретируется как запрос на сокращение емкости по текущему размеру, не обязательный к выполнению.
- При вызове без аргументов функция всегда интерпретируется как запрос на сокращение емкости по текущему размеру, не обязательный к выполнению.
- Емкость никогда не сокращается до величины, меньшей текущего количества символов.
- При каждом перераспределении памяти становятся недействительными все ссылки, указатели и итераторы. Кроме того, перераспределение требует времени, поэтому предварительный вызов `reserve()` повышает скорость работы программы и сохраняет ссылки, указатели и итераторы (подробнее см. с. 469).

## Операции сравнения

```
bool сравнение (const string& str1, const string& str2)
bool сравнение (const string& str, const char* cstr)
bool сравнение (const char* cstr, const string& str)
```

- Первая форма возвращает результат сравнения двух строк.
- Вторая и третья формы возвращают результат сравнения строки с С-строкой.
- Параметр *сравнение* — одна из следующих операций:

```
operator ==
operator !=
operator <
operator >
operator <=
operator >=
```

- Сравнение производится по лексикографическому критерию (см. с. 471).

```
int string::compare (const string& str) const
```

- Сравнивает строку *\*this* со строкой *str*.
- Возвращает:
  - 0, если строки равны;
  - отрицательное число, если *\*this* в лексикографическом отношении меньше *str*;
  - положительное число, если *\*this* в лексикографическом отношении больше *str*.
- Сравнение осуществляется функцией *traits::compare()* (см. с. 660).
- За подробностями обращайтесь к с. 471.

```
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*.
- Если выполняется условие *idx>size()*, генерируется исключение *out\_of\_range()*.
- Сравнение выполняется так же, как в приведенном выше описании *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*, с подстрокой *str*, начинающейся с индекса *str\_idx*.
- Если выполняется условие *idx>size()*, генерируется исключение *out\_of\_range()*.
- Если выполняется условие *str\_idx>str.size()*, генерируется исключение *out\_of\_range()*.
- Сравнение выполняется так же, как в приведенном выше описании *compare(str)*.

```
int string::compare (const char* cstr) const
```

- Сравнивает символы *\*this* с символами C-строки *cstr*.
  - Сравнение выполняется так же, как в приведенном выше описании *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 C-строки *cstr*¹.
 - Сравнение выполняется так же, как в приведенном выше описании *compare(str)*.

```
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 символьного массива *chars*.
- Сравнение выполняется так же, как в приведенном выше описании *compare(str)*.
- Символьный массив *chars* должен содержать не менее *chars_len* символов, которые могут иметь произвольные значения (символ \0 не имеет особой интерпретации).
- Если аргумент *chars_len* равен *string::npos*, генерируется исключение *length_error*.

Обращение к символам

```
char& string::operator[] (size_type idx)
char string::operator[] (size_type idx) const
```

- Обе формы возвращают символ с индексом *idx* (первому символу соответствует индекс 0).
- Для константных строк значение *length()* является действительным индексом; при обращении по нему оператор возвращает значение, сгенерированное конструктором по умолчанию для типа символов (\0 для типа *string*).
- Для неконстантных строк значение *length()* не является действительным индексом.
- Передача недействительного индекса приводит к непредсказуемым последствиям.
- Ссылки, возвращаемые для неконстантных строк, могут стать недействительными из-за модификаций строки или перераспределения памяти (за подробностями обращайтесь к с. 471).
- Если вызывающая сторона не может проверить действительность индекса, следует использовать функцию *at()*.

¹ В стандарте поведение этой формы функции *compare()* определяется иначе: в нем сказано, что *cstr* интерпретируется не как C-строка, а как символьный массив, в качестве длины которого передается значение *pros* (для чего вызывается следующая форма функции *compare()* с передачей *pros* в дополнительном параметре). Это ошибка в стандарте, поскольку такой вызов всегда порождал бы исключение *length_error*.

```
char& string::at (size_type idx)
const char& string::at (size_type idx) const
```

- Обе формы возвращают символ с индексом *idx* (первому символу соответствует индекс 0).
- Для всех строк значение `length()` не является действительным индексом.
- При использовании недействительного индекса (отрицательного, а также большего либо равного `size()`) генерируется исключение `out_of_range`.
- Ссылки, возвращаемые для неконстантных строк, могут стать недействительными из-за модификаций строки или перераспределения памяти (за подробностями обращайтесь к с. 471).
- Если при вызове вы уверены в действительности индекса, можно воспользоваться оператором `[]`, который работает быстрее.

Построение С-строк и символьных массивов

```
const char* string::c_str () const
```

- Возвращает содержимое строки в виде С-строки (массив символов с присоединенным символом `\0`).
- Возвращаемый массив принадлежит строке, поэтомузывающая сторона не должна ни модифицировать, ни удалять возвращаемое значение.
- Возвращаемое значение действительно только на время существования строки и пока для нее вызываются только константные функции.

```
const char* string::data () const
```

- Возвращает содержимое строки в виде символьного массива.
- Возвращаемое значение содержит все символы строки без каких-либо модификаций или дополнений. В частности, символ `\0` не включается в массив. Это означает, что в общем случае возвращаемое значение не является действительной С-строкой.
- Возвращаемый массив принадлежит строке, поэтомузывающая сторона не должна ни модифицировать, ни удалять возвращаемое значение.
- Возвращаемое значение действительно только на время существования строки и пока для нее вызываются только константные функции.

```
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*) в символьный массив *buf*.
- Обе формы возвращают количество скопированных символов.
- Символ `\0` к строке не присоединяется. После вызова содержимое *buf* может не быть действительной С-строкой.

- Перед вызовом необходимо проследить за том, чтобы массив *buf* имел достаточный размер; в противном случае вызов приводит к непредсказуемым последствиям.
- Если выполняется условие *idx*>*size()*, генерируется исключение *out_of_range*.

Модифицирующие операции

Присваивание

string& string::operator= (const string& str)
string& string::assign (const string& str)

- Обе функции присваивают значение строки *str*.
- Обе функции возвращают **this*.

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*.
- Если выполняется условие *str_idx*>*str.size()*, генерируется исключение *out_of_range*.

string& string::operator= (const char cstr)*
string& string::assign (const char cstr)*

- Обе функции присваивают символы, входящие в С-строку *cstr*.
- Присваиваются все символы *cstr*, кроме \0.
- Обе функции возвращают **this*.
- Аргумент *cstr* не должен содержать NULL-указатель.
- Если размер полученной строки превышает максимально допустимое количество символов, обе функции генерируют исключение *length_error*.

string& string::assign (const char chars, size_type chars_len)*

- Присваивает *chars_len* символов из символьного массива *chars*.
- Возвращает **this*.
- Символьный массив *chars* должен содержать не менее *chars_len* символов, которые могут иметь произвольные значения (символ \0 не имеет особой интерпретации).
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение *length_error*.

string& string::operator= (char c)

- Присваивает символ с как новое значение строки.
- Возвращает **this*.
- После вызова **this* содержит только один символ.

```
string& string::assign (size_type num, char c)
```

- Присваивает *num* экземпляров символа *c*.
- Возвращает **this*.
- Если аргумент *num* равен *string::npos*, генерируется исключение *length_error*.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение *length_error*.

```
void string::swap (string& str)
void swap (string& str1, string& str2)
```

- Обе формы меняют местами содержимое двух строк:
 - функция класса обменивает содержимое **this* и *str*,
 - глобальная функция обменивает содержимое *str1* и *str2*.
- По возможности старайтесь использовать эти функции вместо присваивания, поскольку они работают быстрее. Более того, для них гарантируется постоянная сложность. За подробностями обращайтесь к с. 472.

Присоединение символов

```
string& string::operator+= (const string& str)
string& string::append (const string& str)
```

- Обе формы присоединяют к строке символы, входящие в строку *str*.
- Обе формы возвращают **this*.
- Если размер полученной строки превышает максимально допустимое количество символов, обе функции генерируют исключение *length_error*.

```
string& string::append (const string& str, size_type str_idx,
                       size_type str_num)
```

- Присоединяет не более *str_num* символов *str*, начиная с индекса *str_idx*.
- Возвращает **this*.
- Если выполняется условие *str_idx > str.size()*, генерируется исключение *out_of_range*.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение *length_error*.

```
string& string::append (const char* chars, size_type chars_len)
```

- Присоединяет к строке *chars_len* символов символьного массива *chars*.
- Возвращает **this*.
- Символьный массив *chars* должен содержать не менее *chars_len* символов, которые могут иметь произвольные значения (символ *\0* не имеет особой интерпретации).
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение *length_error*.

```
string& string::operator+=(char c)
void string::push_back(char c)
```

- Обе функции присоединяют к строке символ *c*.
- Оператор ***+=*** возвращает ***this**.
- Если размер полученной строки превышает максимально допустимое количество символов, обе функции генерируют исключение **length_error**.

```
string& string::append(InputIterator beg, InputIterator end)
```

- Присоединяет все символы из интервала **[beg, end]**.
- Возвращает ***this**.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение **length_error**.

Вставка символов

```
string& string::insert(size_type idx, const string& str)
```

- Вставляет символы из *str* так, что новые символы начинаются с индекса *idx*.
- Возвращает ***this**.
- Если выполняется условие *idx>size()*, генерируется исключение **out_of_range**.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение **length_error**.

```
string& string::insert(size_type idx, const string& str,
                       size_type str_idx, size_type str_num)
```

- Вставляет не более *str_num* символов из *str*, начиная с индекса *str_idx*, так, что новые символы начинаются с индекса *idx*.
- Возвращает ***this**.
- Если выполняется условие *idx>size()*, генерируется исключение **out_of_range**.
- Если выполняется условие *str_idx>str.size()*, генерируется исключение **out_of_range**.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение **length_error**.

```
string& string::insert(size_type idx, const char* cstr)
```

- Вставляет символы С-строки *cstr* так, что новые символы начинаются с индекса *idx*.
- Возвращает ***this**.
- Аргумент *cstr* не должен содержать NULL-указатель.
- Если выполняется условие *idx>size()*, генерируется исключение **out_of_range**.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение **length_error**.

```
string& string::insert (size_type idx, const char* chars,
                      size_type chars_len)
```

- Вставляет *chars_len* символов из символьного массива *chars* так, что новые символы начинаются с индекса *idx*.
- Возвращает **this*.
- Символьный массив *chars* должен содержать не менее *chars_len* символов, которые могут иметь произвольные значения (символ \0 не имеет особой интерпретации).
- Если выполняется условие *idx>size()*, генерируется исключение *out_of_range*.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение *length_error*.

```
string& string::insert (size_type idx, size_type num, char c)
void string::insert (iterator pos, size_type num, char c)
```

- Обе формы вставляют *num* экземпляров символа *c* в позицию, заданную индексом *idx* или итератором *pos* соответственно.
- Первая форма вставляет новые символы так, чтобы они начинались с индекса *idx*.
- Вторая форма вставляет новые символы перед символом, на который ссылается итератор *pos*.
- Учтите, что перегрузка этих двух функций приводит к потенциальной неоднозначности. При передаче нуля во втором аргументе значение может интерпретироваться как индекс (обычно преобразование к *unsigned*) или как итератор (обычно преобразование к *char**). Следовательно, в таких ситуациях индекс должен передаваться с точным указанием типа. Пример:

```
std::string s;
...
s.insert(0,1,' '); // ОШИБКА: неоднозначность
s.insert((std::string::size_type)0,1,' '); // OK
```

- Обе формы возвращают **this*.
- Если выполняется условие *idx>size()*, обе формы генерируют исключение *out_of_range*.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение *length_error*.

```
iterator string::insert (iterator pos, char c)
```

- Вставляет копию символа *c* перед символом, на который ссылается итератор *pos*.
- Возвращает позицию вставленного символа.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение *length_error*.

```
void string::insert (iterator pos, InputIterator beg, InputIterator end)
```

- Вставляет все символы интервала $[beg, end)$ перед символом, на который ссылается итератор *pos*.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение `length_error`.

Удаление символов

```
void string::clear()
string& string::erase ()
```

- Обе функции удаляют из строки все символы. После вызова строка остается пустой.
- Функция `erase()` возвращает `*this`.

```
string& string::erase (size_type idx)
string& string::erase (size_type idx, size_type len)
```

- Обе формы возвращают не более *len* символов `*this`, начиная с индекса *idx*.
- Обе формы возвращают `*this`.
- Если аргумент *len* отсутствует, удаляются все оставшиеся символы.
- Если выполняется условие $idx > \text{size}()$, обе формы генерируют исключение `out_of_range`.

```
string& string::erase (iterator pos)
string& string::erase (iterator beg, iterator end)
```

- Обе формы удаляют один символ в позиции итератора *pos* или все символы в интервале $[beg, end)$ соответственно.
- Обе формы возвращают первый символ после удаленного фрагмента (вторая форма возвращает *end*)¹.

Изменение размера

```
void string::resize (size_type num)
void string::resize (size_type num, char c)
```

- Обе формы изменяют количество символов в `*this` и делают его равным *num*. Если значение *num* отлично от `size()`, функции присоединяют или удаляют символы в конце строки в соответствии с новым размером.
- При увеличении количества символов новые символы инициализируются значением *c*. Если аргумент *c* отсутствует, символы инициализируются конструктором по умолчанию для типа символов (то есть `\0` для типа `char`).
- Если аргумент *num* равен `string::npos`, обе формы генерируют исключение `length_error`.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение `length_error`.

¹ В стандарте допущена ошибка — в нем говорится, что вторая форма возвращает позицию за *end*.

Замена символов

```
string& string::replace (size_type idx, size_type len, const string& str)
string& string::replace (iterator beg, iterator end, const string& str)
```

- Первая форма заменяет не более *len* символов **this*, начиная с индекса *idx*, всеми символами строки *str*.
- Вторая форма заменяет все символы в интервале *[beg,end)* всеми символами *str*.
- Обе формы возвращают **this*.
- Если выполняется условие *idx>size()*, обе формы генерируют исключение *out_of_range*.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение *length_error*.

```
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 **this*, начиная с индекса *idx*, не более чем *str_num* символами строки *str*, начиная с индекса *str_idx*.
- Возвращает **this*.
- Если выполняется условие *idx>size()*, обе формы генерируют исключение *out_of_range*.
- Если выполняется условие *str_idx>str.size()*, обе формы генерируют исключение *out_of_range*.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение *length_error*.

```
string& string::replace (size_type idx, size_type len, const char* cstr)
string& string::replace (iterator beg, iterator end, const char* cstr)
```

- Обе формы заменяют не более *len* символов **this*, начиная с индекса *idx*, или все символы интервала *[beg,end)* соответственно всеми символами С-строки *cstr*.
- Обе формы возвращают **this*.
- Аргумент *cstr* не должен содержать NULL-указатель.
- Если выполняется условие *idx>size()*, обе формы генерируют исключение *out_of_range*.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение *length_error*.

```
string& string::replace (size_type idx, size_type len,
                        const char* chars, size_type chars_len)
string& string::replace (iterator beg, iterator end,
                        const char* chars, size_type chars_len)
```

- Обе формы заменяют не более *len* символов **this*, начиная с индекса *idx*, или все символы интервала *[beg,end)* соответственно *chars_len* символами массива *chars*.

- Обе формы возвращают `*this`.
- Символьный массив `chars` должен содержать не менее `chars_len` символов, которые могут иметь произвольные значения (символ `\0` не имеет особой интерпретации).
- Если выполняется условие `idx>size()`, обе формы генерируют исключение `out_of_range`.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение `length_error`.

```
string& string::replace (size_type idx, size_type len,
                        size_type num, char c)
string& string::replace (iterator beg, iterator end,
                        size_type num, char c)
```

- Обе формы заменяют не более `len` символов `*this`, начиная с индекса `idx`, или все символы интервала `[beg,end)` соответственно `num` экземплярами символа `c`.
- Обе формы возвращают `*this`.
- Если выполняется условие `idx>size()`, обе формы генерируют исключение `out_of_range`.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение `length_error`.

```
string& string::replace (iterator beg, iterator end,
                        InputIterator newBeg, InputIterator newEnd)
```

- Заменяет все символы интервала `[beg,end)` всеми символами интервала `[newBeg, newEnd)`.
- Возвращает `*this`.
- Если размер полученной строки превышает максимально допустимое количество символов, генерируется исключение `length_error`.

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 idx) const
```

- Функции ищут первое/последнее вхождение символа `c` (начиная с индекса `idx`).
- Функция `find()` выполняет поиск в прямом направлении и находит первое вхождение.
- Функция `rfind()` выполняет поиск в обратном направлении и находит последнее вхождение.
- Все функции возвращают индекс символа при успешном поиске или `string::npos()` в случае неудачи.

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 idx) const
```

- Функции ищут первое/последнее вхождение подстроки *str* (начиная с индекса *idx*).
- Функция *find()* выполняет поиск в прямом направлении и находит первое вхождение подстроки.
- Функция *rfind()* выполняет поиск в обратном направлении и находит последнее вхождение подстроки.
- Все функции возвращают индекс первого символа подстроки при успешном поиске или *string::npos()* в случае неудачи.

```
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 idx) const
```

- Функции ищут первое/последнее вхождение подстроки, содержащей символы С-строки *cstr* (начиная с индекса *idx*).
- Функция *find()* выполняет поиск в прямом направлении и находит первое вхождение подстроки.
- Функция *rfind()* выполняет поиск в обратном направлении и находит последнее вхождение подстроки.
- Все функции возвращают индекс первого символа подстроки при успешном поиске или *string::npos()* в случае неудачи.
- Аргумент *cstr* не должен содержать NULL-указатель.

```
size_type string::find (const char* chars, size_type idx,  
                      size_type chars_len) const  
size_type string::rfind (const char* chars, size_type idx,  
                      size_type chars_len) const
```

- Функции ищут первое/последнее вхождение подстроки, содержащей *chars_len* символов массива *chars* (начиная с индекса *idx*).
- Функция *find()* выполняет поиск в прямом направлении и находит первое вхождение подстроки.
- Функция *rfind()* выполняет поиск в обратном направлении и находит последнее вхождение подстроки.
- Обе функции возвращают индекс первого символа подстроки при успешном поиске или *string::npos()* в случае неудачи.
- Символьный массив *chars* должен содержать не менее *chars_len* символов, которые могут иметь произвольные значения (символ \0 не имеет особой интерпретации).

Поиск первого вхождения одного из нескольких 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
```

- Функции ищут первый символ, который также входит или не входит в строку *str* (начиная с индекса *idx*).
- Функции возвращают индекс символа при успешном поиске или *string::npos()* в случае неудачи.

```
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
```

- Функции ищут первый символ, который также входит или не входит в С-строку *cstr* (начиная с индекса *idx*).
- Функции возвращают индекс символа при успешном поиске или *string::npos()* в случае неудачи.
- Аргумент *cstr* не должен содержать NULL-указатель.

```
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
```

- Функции ищут первый символ, который также входит или не входит в *chars_len* символов символьного массива *chars* (начиная с индекса *idx*).
- Функции возвращают индекс символа при успешном поиске или *string::npos()* в случае неудачи.
- Символьный массив *chars* должен содержать не менее *chars_len* символов, которые могут иметь произвольные значения (символ \0 не имеет особой интерпретации).

```
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
```

- Функции ищут первый символ, значение которого равно или не равно *c* (начиная с индекса *idx*).
- Функции возвращают индекс символа при успешном поиске или *string::npos()* в случае неудачи.

Поиск последнего вхождения одного из нескольких символов

```
size_type string::find_last_of (const string& str) const
size_type string::find_last_of (const string& str, size_type idx) const
```

```
size_type string::find_last_not_of (const string& str) const
size_type string::find_last_not_of (const string& str, size_type idx) const
```

- Функции ищут последний символ, который также входит или не входит в строку *str* (начиная с индекса *idx*).
- Функции возвращают индекс символа при успешном поиске или *string::npos()* в случае неудачи.

```
size_type string::find_last_of (const char* cstr) const
size_type string::find_last_of (const char* cstr, size_type idx) const
size_type string::find_last_not_of (const char* cstr) const
size_type string::find_last_not_of (const char* cstr, size_type idx) const
```

- Функции ищут последний символ, который также входит или не входит в С-строку *cstr* (начиная с индекса *idx*).
- Функции возвращают индекс символа при успешном поиске или *string::npos()* в случае неудачи.
- Аргумент *cstr* не должен содержать NULL-указатель.

```
size_type string::find_last_of (const char* chars, size_type idx,
                               size_type chars_len) const
size_type string::find_last_not_of (const char* chars, size_type idx,
                                   size_type chars_len) const
```

- Функции ищут последний символ, который также входит или не входит в *chars_len* символов символьного массива *chars* (начиная с индекса *idx*).
- Функции возвращают индекс символа при успешном поиске или *string::npos()* в случае неудачи.
- Символьный массив *chars* должен содержать не менее *chars_len* символов, которые могут иметь произвольные значения (символ \0 не имеет особой интерпретации).

```
size_type string::find_last_of (char c) const
size_type string::find_last_of (char c, size_type idx) const
size_type string::find_last_not_of (char c) const
size_type string::find_last_not_of (char c, size_type idx) const
```

- Функции ищут последний символ, значение которого равно или не равно *c* (начиная с индекса *idx*).
- Функции возвращают индекс символа при успешном поиске или *string::npos()* в случае неудачи.

Выделение подстрок и конкатенация

```
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 из не более чем *len* символов строки **this*, начиная с индекса *idx*.

- Если аргумент *len* отсутствует, используются все оставшиеся символы строки.
- Если аргументы *idx* и *len* отсутствуют, возвращается копия строки.
- Если выполняется условие *idx*>*size()*, все три формы генерируют исключение *out_of_range*.

```
string operator+ (const string& str1, const string& str2)
string operator+ (const string& str1, const char* cstr)
string operator+ (const char* cstr, const string& str)
string operator+ (const string& str, char c)
string operator+ (char c, const string& str)
```

- Все формы выполняют конкатенацию двух операндов и возвращают полученную строку.
- Операнды состоят из произвольной комбинации строк, С-строк или одиночных символов.
- Если размер полученной строки превышает максимально допустимое количество символов, все формы генерируют исключение *length_error*.

ФУНКЦИИ ВВОДА-ВЫВОДА

ostream& operator<< (ostream& strm, const string& str)

- Записывает символы *str* в поток *strm*.
- Если выполняется условие *strm.width()>0*, в поток данных выводятся не менее *width()* символов, а ширина поля в потоке данных обнуляется.
- Параметр *ostream* — выходной поток типа *basic_ostream<char>*, где *char* — тип символов (см. с. 562).

istream& operator>> (istream& strm, string& str)

- Читает символы следующего слова из потока данных *strm* в строку *str*.
- Если для *strm* установлен флаг *skipws*, начальные пропуски игнорируются.
- Чтение производится до выполнения одного из следующих условий:
 - *strm.width()* больше 0, и из потока данных прочитаны *width()* символов;
 - *strm.good()* возвращает *false* (что может привести к выдаче соответствующего исключения);
 - *isspace(c,strm.getloc())* возвращает *true* для следующего символа *c*;
 - *str.max_size()* символов сохраняется.
- Параметр *istream* — входной поток данных типа *basic_istream<char>*, где *char* — тип символов (см. с. 562).

istream& getline (istream& strm, string& str)

istream& getline (istream& strm, string& str, char delim)

- Читает символы следующей логической строки из потока *strm* в строку *str*.

- Читаются все символы (включая начальные пропуски) до выполнения одного из следующих условий:
 - *strm.good()* возвращает *false* (что может привести к выдаче соответствующего исключения);
 - из потока данных читается *delim* или *strm.widen('\n')*;
 - *str.max_size()* символов сохраняется.
- Разделитель строк извлекается из потока данных, но не присоединяется к *str*.
- Внутренняя память перераспределяется по мере необходимости.
- Параметр *istream* — входной поток данных типа *basic_istream<char>*, где *char* — тип символов (см. с. 562).

Получение итераторов

iterator string::begin ()
const_iterator string::begin () const

- Обе формы возвращают итератор произвольного доступа, установленный в начало строки (в позицию первого символа).
- Для пустой строки вызов *begin()* эквивалентен вызову *end()*.

iterator string::end ()
const_iterator string::end () const

- Обе формы возвращают итератор произвольного доступа, установленный в конец строки (в позицию за последним символом).
- Символ в позиции *end()* не определен, поэтому конструкции вида **s.end()* приводят к непредсказуемым последствиям.
- Для пустой строки вызов *end()* эквивалентен вызову *begin()*.

reverse_iterator string::rbegin ()
const_reverse_iterator string::rbegin () const

- Обе формы возвращают итератор произвольного доступа, установленный в позицию начала перебора строки в обратном направлении (то есть в позицию последнего символа).
- Для пустой строки вызов *rbegin()* эквивалентен вызову *rend()*.
- Обратные итераторы описаны на с. 270

reverse_iterator string::rend ()
const_reverse_iterator string::rend () const

- Обе формы возвращают итератор произвольного доступа, установленный в позицию конца перебора строки в обратном направлении (в позицию перед первым символом).
- Символ в позиции *rend()* не определен, поэтому конструкции вида **s.rend()* приводят к непредсказуемым последствиям.

- Для пустой строки вызов `rend()` эквивалентен вызову `rbegin()`.
- Обратные итераторы описаны на с. 270

Поддержка распределителей памяти

Для строк также определены стандартные члены классов, обеспечивающие поддержку распределителей памяти.

`string::allocator_type`

- Тип распределителя памяти.
- Третий параметр шаблона `basic_string<>`.
- Для типа `string` — эквивалент `allocator<char>`.

`allocator_type string::get_allocator () const`

Возвращает модель распределения памяти, используемую строкой.

Распределитель памяти может также передаваться в необязательном аргументе всех конструкторов строк. Ниже перечислены строковые конструкторы с необязательными аргументами в соответствии со стандартом:

```
namespace std {
    template<class charT,
              class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
    class basic_string {
        public:
            // Конструктор по умолчанию
            explicit basic_string(const Allocator& a = Allocator());

            // Копирующий конструктор
            basic_string(const basic_string& str,
                         size_type str_idx = 0,
                         size_type str_num = npos);
            basic_string(const basic_string& str,
                         size_type str_idx, size_type str_num,
                         const Allocator&);

            // Конструктор для C-строк
            basic_string(const charT* cstr,
                         const Allocator& a = Allocator());

            // Конструктор для символьных массивов
            basic_string(const charT* chars, size_type chars_len,
                         const Allocator& a = Allocator());

            // Конструктор для пир экземпляров символа
            basic_string(size_type num, charT c,
                         const Allocator& a = Allocator());
    };
}
```

```
// Конструктор для интервала символов
template<class InputIterator>
basic_string(InputIterator beg, InputIterator end,
            const Allocator& a = Allocator());
    ...
}:
```

Эти конструкторы ведут себя так, как описано на с. 490, но дополнительно позволяют передать пользовательский объект распределителя памяти. Если строка инициализируется другой строкой, то вместе с содержимым копируется и распределитель памяти¹. За дополнительной информацией о распределителях памяти обращайтесь к главе 15.

¹ В стандарте сказано, что при копировании строки используется распределитель памяти по умолчанию. Такое решение выглядит нелогично, поэтому в настоящем времени рассматривается предложение об изменении этой формулировки.

12 Числовые типы

В этой главе описаны компоненты стандартной библиотеки C++, предназначенные для работы с числовыми данными. В частности, здесь будет представлен класс комплексных чисел, классы массивов значений и глобальные численные функции, унаследованные из библиотеки C.

Еще два численных компонента стандартной библиотеки C++ описаны в других главах:

- несколько численных алгоритмов, поддерживаемых библиотекой STL, представлены на с. 414;
- для всех базовых типов данных некоторые аспекты представления зависят от реализации и описываются типом `numeric_limits`, который рассматривается на с. 72.

Комплексные числа

В стандартную библиотеку C++ входит шаблонный класс `complex<>`, предназначенный для работы с комплексными числами. На всякий случай стоит напомнить, что комплексные числа состоят из двух частей: вещественной и мнимой. Особое свойство мнимой части заключается в том, что ее квадрат является отрицательным числом. Иначе говоря, мнимая часть представляет собой произведение числа и квадратного корня из -1 , обозначаемого символом i .

Класс `complex` объявляется в заголовочном файле `<complex>`:

```
#include <complex>
```

Определение класса `complex` в файле `<complex>` выглядит так:

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

Параметр шаблона `T` задает скалярный тип как вещественной, так и мнимой части комплексного числа.

В стандартную библиотеку C++ включены также три специализированные версии класса `complex` для типов `float`, `double` и `long double`:

```
namespace std {
    template<> class complex<float>;
    template<> class complex<double>;
    template<> class complex<long double>;
}
```

Определения этих типов позволяют выполнять некоторые виды оптимизации и безопасное преобразование от одного комплексного типа к другому.

Примеры использования класса `complex`

Следующая программа демонстрирует возможности класса `complex` по созданию комплексных чисел, их выводу в разных представлениях и выполнению некоторых типовых операций с комплексными числами.

```
// num/complex1.cpp
#include <iostream>
#include <complex>
using namespace std;

int main()
{
    /* Комплексное число с вещественной и мнимой частями
     * - Вещественная часть: 4.0
     * - Мнимая часть: 3.0
     */
    complex<double> c1(4.0,3.0);

    /* Создание комплексного числа в системе полярных координат
     * - Амплитуда: 5.0
     * - Фазовый угол: 0.75
     */
    complex<float> c2(polar(5.0,0.75));

    // Вывод комплексного числа с вещественной и мнимой частями
    cout << "c1: " << c1 << endl;
    cout << "c2: " << c2 << endl;

    // Вывод комплексного 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;

    // Вывод сопряжений комплексных чисел
    cout << "c1 conjugated: " << conj(c1) << endl;
    cout << "c2 conjugated: " << conj(c2) << endl;
```

```

// Вывод результата вычисления
cout << "4.4 + c1 * 1.8: " << 4.4 + c1 * 1.8 << endl;

/* Вывод суммы c1 и c2:
 * - внимание: разные типы!
 */
cout << "c1 + c2:      "
     << c1 + complex<double>(c2.real(),c2.imag()) << endl;

// Прибавление к c1 квадратного корня из c1 и вывод результата
cout << "c1 += sqrt(c1): " << (c1 += sqrt(c1)) << endl;
}

```

Примерный результат выполнения программы выглядит так (точный результат зависит от реализации некоторых свойств 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)

```

Вторая программа в цикле читает два комплексных числа и вычисляет результат возведения первого числа в степень второго:

```

// num/complex2.cpp
#include <iostream>
#include <complex>
#include <cstdlib>
#include <climits>
using namespace std;

int main()
{
    complex<long double> c1, c2;

    while (cin.peek() != EOF) {

        // Ввод первого комплексного числа
        cout << "complex number c1: ";
        cin >> c1;
        if (!cin) {
            cerr << "input error" << endl;
            return EXIT_FAILURE;
        }

        // Ввод второго комплексного числа
        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;

    // Пропуск оставшейся части строки
    cin.ignore(numeric_limits<int>::max(),'\'n');
}
}

```

В табл. 12.1 приведены примеры входных данных и полученных результатов.

Таблица 12.1. Возможные результаты выполнения программы complex2.cpp

c1	c2	Результат
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: (-4,4.8984e-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)

Обратите внимание: при вводе комплексного числа указывается либо только вещественная часть в виде отдельного значения (в круглых скобках или без), либо вещественная и мнимая части в круглых скобках, разделенные запятыми.

Операции с комплексными числами

Далее описаны операции с комплексными числами, поддерживаемые классом `complex`.

Создание, копирование и присваивание комплексных чисел

В табл. 12.2 перечислены конструкторы и операции присваивания для типа `complex`. Конструкторам могут передаваться исходные значения вещественной и мнимой частей. Если значения не заданы, они инициализируются конструктором по умолчанию для соответствующего типа.

Таблица 12.2. Конструкторы и операции присваивания класса `complex<>`

Выражение	Эффект
complex c	Создает комплексное число с нулевой вещественной и мнимой частями ($0+0i$)
complex c(1.3)	Создает комплексное число с вещественной частью 1.3 и нулевой мнимой частью ($1.3+0i$)
complex c(1.3, 2.4)	Создает комплексное число с вещественной частью 1.3 и мнимой частью 4.2 ($1.3+4.2i$)
complex c1(c2)	Создает комплексное число c1 как копию c2
polar(4.2)	Создает временное комплексное число по полярным координатам (амплитуда $\rho = 4.2$, фазовый угол $\phi = 0$)
polar(4.2, 0.75)	Создает временное комплексное число по полярным координатам (амплитуда $\rho = 4.2$, фазовый угол $\phi = 0.75$)
conj(c)	Создает временное комплексное число, сопряженное с числом c (то есть комплексное число с противоположным знаком мнимой части)
c1 = c2	Присваивает c1 вещественную и мнимую части c2
c1 += c2	Прибавляет c2 к c1
c1 -= c2	Вычитает c2 из c1
c1 *= c2	Умножает c1 на c2
c1 /= c2	Делит c1 на c2

Значение существующего комплексного числа может быть изменено только при помощи операторов присваивания. Комбинированные операторы присваивания `+=`, `-=`, `*=` и `/=` осуществляют суммирование, вычитание, умножение и деление значений двух комплексных operandов.

Вспомогательная функция `polar()` позволяет создать комплексное число, инициализируемое в полярных координатах (через амплитуду и фазовый угол, указанный в радианах):

```
// Создание комплексного числа с инициализацией в полярных координатах  
std::complex<double> c2(std::polar(4.2,0.75));
```

Если в создании комплексного числа задействовано неявное преобразование типа, возникает проблема. Например, следующее решение работает нормально:

```
std::complex<float> c2(std::polar(4.2,0.75)); // OK
```

Однако похожая запись со знаком равенства ошибочна:

```
std::complex<float> c2 = std::polar(4.2,0.75); // ОШИБКА
```

Эта проблема рассматривается в далее.

Вспомогательная функция `conj()` позволяет создать комплексное число, инициализированное значением, сопряженным с другим комплексным числом (то есть комплексным числом с противоположным знаком мнимой части):

```
std::complex<double> c1(1.1,5.5);
```

```
std::complex<double> c2(conj(c1)); // c2 инициализируется как  
// complex<double>(1.1, -5.5)
```

Неявные преобразования типов

Конструкторы специализированных версий для типов `float`, `double` и `long double` спроектированы так, чтобы безопасные преобразования типов (например, `complex<float>` в `complex<double>`) могли выполняться неявно, а потенциально рискованные преобразования (например, `complex<long double>` в `complex<double>`) были явными (см. с. 521):

```
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: нет неявного преобразования
std::complex<double> cd3(cld); // OK: явное преобразование
```

Не существует конструкторов, создающих комплексное число по другим комплексным типам. В частности, нельзя преобразовать `complex` с целым типом в `complex` с типом `float`, `double` или `long double`. Впрочем, преобразования можно выполнять при передаче вещественной и мнимой частей в отдельных аргументах:

```
std::complex<double> cd;
std::complex<int> ci;
...
std::complex<double> cd4 = ci; // ОШИБКА: небезопасное преобразование
std::complex<double> cd5(ci); // ОШИБКА: нет явного преобразования
std::complex<double> cd6(ci.real(), ci.imag()); // OK
```

К сожалению, операторы присваивания позволяют выполнять небезопасные преобразования. Они определены в виде шаблонов для всех типов, поэтому присваивание допустимо для любых комплексных типов (при возможности преобразования типа значения)¹:

```
std::complex<double> cd;
std::complex<long double> cld;
std::complex<int> ci;
...
cd = ci; // OK
cd = cld; // OK
```

Эта проблема также относится к функциям `polar()` и `conj()`. Например, следующая запись работает нормально:

```
std::complex<float> c2(std::polar(4.2, 0.75)); // OK
```

С другой стороны, запись со знаком `=` не работает:

```
std::complex<float> c2 = std::polar(4.2, 0.75); // ОШИБКА
```

¹ Вероятно, тот факт, что конструкторы допускают только безопасные неявные преобразования типов, тогда как операции присваивания допускают любые неявные преобразования, является ошибкой стандарта.

Дело в том, что выражение `std::polar(4.2,0.75)` создает временный объект `complex<double>`, а неявное преобразование из `complex<double>` в `complex<float>` не определено¹.

Доступ к данным

В табл. 12.3 перечислены функции получения атрибутов комплексных чисел.

Таблица 12.3. Операции доступа к данным класса `complex<>`

Выражение	Описание
<code>c.real()</code>	Возвращает значение вещественной части (функция класса)
<code>real(c)</code>	Возвращает значение вещественной части (глобальная функция)
<code>c.imag()</code>	Возвращает значение мнимой части (функция класса)
<code>imag(c)</code>	Возвращает значение мнимой части (глобальная функция)
<code>abs(c)</code>	Возвращает модуль с ($\sqrt{c.\text{real}()^2 + c.\text{imag}()^2}$)
<code>norm(c)</code>	Возвращает квадрат модуля с ($c.\text{real}()^2 + c.\text{imag}()^2$)
<code>arg(c)</code>	Возвращает фазовый угол в полярном представлении с (ϕ); эквивалент <code>atan2(c.imag(),c.real())</code>

Функции `real()` и `imag()` позволяют только прочитать значения вещественной и мнимой частей. Чтобы изменить любую из частей комплексного числа, необходимо присвоить ему новое значение. Например, следующая команда присваивает мнимой части с значение 3.7:

```
std::complex<double> c;
...
c = std::complex<double>(c.real(),3.7);
```

Операции сравнения

Из всех операций сравнения для комплексных чисел определены только проверки на равенство и на неравенство (табл. 12.4). Операторы `==` и `!=` определены как глобальные функции, поэтому один из операндов может быть скалярной величиной. В этом случае operand интерпретируется как вещественная часть, а мнимой части комплексного числа присваивается значение по умолчанию для данного типа (обычно 0).

Другие операции сравнения (например, с оператором `<` и т. д.) для класса `complex` не определены. Хотя в принципе для комплексных чисел можно определить порядок сортировки, результат получается недостаточно интуитивным и не приносит особой практической пользы. Например, сравнивать комплексные числа

¹ Между следующими двумя фрагментами существует нетривиальное различие:

```
X x;
Y y(x); // Явное преобразование
X x;
Y y = x; // Неявное преобразование
```

на основании модулей бессмысленно, поскольку два разных комплексных числа (например, 1 и -1) могут иметь одинаковые модули. Конечно, можно изобрести специальный критерий сортировки, например, для двух комплексных чисел $c1$ и $c2$ считать, что $c1 < c2$ при выполнении условия $|c1| < |c2|$, а в случае совпадения модулей — при выполнении условия $\arg(c1) < \arg(c2)$. Тем не менее такие искусственные критерии не обладают математическим смыслом¹.

Таблица 12.4. Операции сравнения для класса `complex<>`

Выражение	Описание
<code>c1 == c2</code>	Проверка на равенство $c1$ и $c2$ ($c1.real() == c2.real()$ && $c1.imag() == c2.imag()$)
<code>c == 1.7</code>	Проверка на равенство $c1$ и 1.7 ($c1.real() == 1.7$ && $c1.imag() == 0.0$)
<code>1.7 == c</code>	Проверка на равенство 1.7 и c ($c1.real() == 1.7$ && $c1.imag() == 0.0$)
<code>c1 != c2</code>	Проверка на неравенство $c1$ и $c2$ ($c1.real() != c2.real()$ $c1.imag() != c2.imag()$)
<code>c != 1.7</code>	Проверка на неравенство $c1$ и 1.7 ($c1.real() != 1.7$ $c1.imag() != 0.0$)
<code>1.7 != c</code>	Проверка на неравенство 1.7 и c ($c1.real() != 1.7$ $c1.imag() != 0.0$)

Из этого следует, что тип `complex` не может быть типом элементов ассоциативных контейнеров (без определения пользовательского критерия сортировки). Дело в том, что для сортировки элементов по умолчанию ассоциативные контейнеры используют объект функции `less<>`, который вызывает оператор `<` (см. с. 143).

Определение пользовательского оператора `<` позволяет сортировать комплексные числа и использовать их в ассоциативных контейнерах. Будьте внимательны и не нарушайте стандартное пространство имен. Пример:

```
template <class 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 базовые арифметические операции, а также операции изменения знака (табл. 12.5).

Таблица 12.5. Арифметические операции класса `complex<>`

Выражение	Описание
<code>c1 + c2</code>	Возвращает сумму $c1$ и $c2$
<code>c + 1.7</code>	Возвращает сумму c и 1.7

продолжение ↓

¹ Спасибо Дэвиду Вандеворду (David Vandevoorde) за это пояснение.

Таблица 12.5 (продолжение)

Выражение	Описание
$1.7 + c$	Возвращает сумму 1.7 и с
$c1 - c2$	Возвращает разность c1 и c2
$c - 1.7$	Возвращает разность с и 1.7
$1.7 - c$	Возвращает разность 1.7 и с
$c1 * c2$	Возвращает произведение c1 и c2
$c * 1.7$	Возвращает произведение с и 1.7
$1.7 * c$	Возвращает произведение 1.7 и с
$c1 / c2$	Возвращает частное от деления c1 на c2
$c / 1.7$	Возвращает частное от деления с на 1.7
$1.7 / c$	Возвращает частное от деления 1.7 на с
$- c$	Возвращает значение с с обратным знаком
$+ c$	Возвращает с
$c1 += c2$	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

В классе `complex` определены стандартные операторы ввода-вывода `<<` и `>>` (табл. 12.6).

Таблица 12.6. Операции ввода-вывода класса `complex<>`

Выражение	Описание
<code>strm << c</code>	Записывает комплексное число с в выходной поток данных strm
<code>strm >> c</code>	Читает комплексное число с из входного потока данных strm

Оператор `>>` выводит комплексное число в поток данных в следующем формате:
`(вещественная_часть , мнимая_часть)`

Эквивалентная реализация оператора вывода выглядит так:

```
template <class T, class charT, class traits>
std::basic_ostream<charT,traits>&
operator<< (std::basic_ostream<charT,traits>& strm,
           const std::complex<T> c)
{
    // Временная строка для выполнения вывода с одним аргументом
    std::basic_ostringstream<charT,traits> s;
```

```

s.flags(strm.flags());           // Копирование флагов потока
s.imbue(strm.getloc());         // Копирование локального контекста потока
s.precision(strm.precision());   // Копирование точности потока

// Подготовка итоговой строки
s << '(' << c.real() << ',' << c.imag() << ')';

// Вывод итоговой строки
strm << s.str();

return strm;
}

```

Оператор ввода читает комплексные числа в одном из представленных ниже форматов:

(вещественная_часть, мнимая_часть)
(вещественная_часть)
вещественная_часть

Если очередные символы во входном потоке данных не соответствуют ни одному из перечисленных форматов, устанавливается флаг `ios::failbit`, что может привести к соответствующему исключению (см. с. 576).

К сожалению, вы не можете задать собственный разделитель для вещественной и мнимой частей комплексного числа. А это означает, что при использовании запятой в качестве «десятичной точки» (как, например, в Германии или России) вводимые/выводимые данные выглядят довольно странно. Например, комплексное число с вещественной частью 4.6 и мнимой частью 2.7 записывается в виде:
(4.6,2.7)

Пример использования операций ввода-вывода приведен на с. 512.

Трансцендентные функции

В табл. 12.7 перечислены трансцендентные функции (тригонометрия, возведение в степень и т. д.) класса `complex`.

Таблица 12.7. Трансцендентные функции класса `complex<>`

Выражение	Описание
<code>pow(c,3)</code>	Комплексное возведение в степень: c^3
<code>pow(c,1.7)</code>	Комплексное возведение в степень: $c^{1.7}$
<code>pow(c1,c2)</code>	Комплексное возведение в степень: $c1^{c2}$
<code>pow(1.7,c)</code>	Комплексное возведение в степень: 1.7^c
<code>exp(c)</code>	Возведение в степень по основанию е: e^c
<code>sqrt(c)</code>	Квадратный корень из с (\sqrt{c})
<code>log(c)</code>	Комплексный натуральный логарифм с ($\ln c$)
<code>log10(c)</code>	Комплексный десятичный логарифм с ($\lg c$)

продолжение ↴

Таблица 12.7 (продолжение)

Выражение	Описание
<code>sin(c)</code>	Синус с ($\sin c$)
<code>cos(c)</code>	Косинус с ($\cos c$)
<code>tan(c)</code>	Тангенс с ($\tan c$)
<code>sinh(c)</code>	Гиперболический синус с ($\sinh c$)
<code>cosh(c)</code>	Гиперболический косинус с ($\cosh c$)
<code>tanh(c)</code>	Гиперболический тангенс с ($\tanh c$)

Строение класса `complex`

Здесь приведены подробные описания всех операций класса `complex<T>`. Обозначение `T` задает параметр шаблона класса `complex<T>`, то есть тип вещественной и мнимой частей значения `complex`.

Определения типов

`complex::value_type`

Тип вещественной и мнимой частей.

Операции создания, копирования и присваивания

`complex::complex ()`

- Конструктор по умолчанию.
- Создает комплексное значение, в котором вещественная и мнимая части инициализируются вызовом конструкторов по умолчанию (0 для базовых типов данных). Это означает, что для базовых типов данных исходные значения вещественной и мнимой частей равны 0 (инициализация по умолчанию для базовых типов описана на с. 30).

`complex::complex (const T& re)`

- Создает комплексное значение, в котором вещественная часть равна `re`, а мнимая часть инициализируется вызовом конструктора по умолчанию (0 для базовых типов данных).
- Конструктор также определяет автоматическое преобразование типа `T` в `complex`.

`complex::complex (const T& re, const T& im)`

Создает комплексное значение, в котором вещественная и мнимая части равны соответственно `re` и `im`.

`complex polar (const T& rho)`

`complex polar (const T& rho, const T& theta)`

- Обе формы создают и возвращают комплексное число, инициализированное в полярных координатах.

- Параметр *rho* определяет амплитуду.
- Параметр *theta* определяет фазовый угол в радианах (по умолчанию 0).

complex conj (const complex& cmplx)

Создает и возвращает комплексное число, инициализируемое сопряжением другого комплексного числа (то есть числом с противоположным знаком мнимой части).

complex::complex (const complex& cmplx)

- Копирующий конструктор.
- Создает новое комплексное число как копию *cmplx*.
- Копирует вещественную и мнимую части.
- Функция определяется в обеих формах: шаблонной и обычной (шаблонные функции классов описаны на с. 28). Таким образом, конструктор обеспечивает автоматическое преобразование к типу элементов.
- В специализированных версиях для типов *float*, *double* и *long double* копирующие конструкторы ограничиваются так, чтобы небезопасные преобразования (*double* и *long double* в *float*, *long double* в *double*) выполнялись явно, а другие преобразования к типу элементов были запрещены:

```
namespace std {
    template<> class complex<float> {
        public:
            explicit complex(const complex<double>&);
            explicit complex(const complex<long double>&);
            // Другие версии копирующих конструкторов отсутствуют
            ...
    };
    template<> class complex<double> {
        public:
            complex(const complex<float>&);
            explicit complex(const complex<long double>&);
            // Другие версии копирующих конструкторов отсутствуют
            ...
    };
    template<> class complex<long double> {
        public:
            complex(const complex<float>&);
            complex(const complex<double>&);
            // Другие версии копирующих конструкторов отсутствуют
            ...
    };
}
```

На с. 515 приведена дополнительная информация по этой теме.

complex& complex::operator= (const complex& cmplx)

- Присваивает значение комплексного числа *cmplx*.
- Возвращает **this*.

- Функция определяется в обеих формах: шаблонной и обычной (шаблонные функции классов описаны на с. 28). Тем самым обеспечивается автоматическое преобразование к типу элементов (это относится и к специализированным версиям из стандартной библиотеки 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)
```

- Сложение, вычитание, умножение и деление *cmplx* и **this* с сохранением результата в **this*.
- Операторы возвращают **this*.
- Операции определяются в обеих формах: как шаблонные и обычные функции (шаблонные функции классов описаны на с. 28). Тем самым обеспечивается автоматическое преобразование к типу элементов (это относится и к специализированным версиям из стандартной библиотеки C++).

Обратите внимание: операторы присваивания — единственные функции, позволяющие изменить значение существующего объекта *complex*.

Обращение к данным

```
T complex::real () const
T real (const complex& cmplx)
T complex::imag () const
T imag (const complex& cmplx)
```

- Функции возвращают значение вещественной или мнимой части соответственно.
- Возвращаемое значение не является ссылкой. Это означает, что функции *real()* и *imag()* не могут использоваться для изменения вещественной или мнимой части числа. Чтобы изменить только вещественную или мнимую часть, необходимо присвоить объекту *complex* новое комплексное число (см. с. 516).

T abs (const complex& cmplx)

- Возвращает модуль (абсолютное значение) комплексного числа *cmplx*.
- По формуле $\sqrt{cmplx.real()^2 + cmplx.imag()^2}$ вычисляется модуль комплексного числа.

T norm (const complex& cmplx)

- Возвращает квадрат модуля комплексного числа *cmplx*.
- Квадрат модуля комплексного числа вычисляется по формуле *cmplx.real()^2 + cmplx.imag()^2*.

T arg (const complex& cmplx)

- Возвращает фазовый угол представления *cmplx* в полярных координатах (ϕ).
- Эквивалент:

atan2(cmplx.imag(), cmplx.real()).

Операции ввода-вывода

`ostream& operator<< (ostream& strm, const complex& cmplx)`

- Выводит значение `cmplx` в поток данных `strm` в формате:

(вещественная_часть, мнимая_часть)

- Возвращает `strm`.

- Поведение операторов ввода и вывода более подробно описано на с. 518.

`istream& operator>> (istream& strm, complex& cmplx)`

- Читает новое значение из потока данных `strm` в переменную `cmplx`.

- Допустимые форматы входных данных:

(вещественная_часть, мнимая_часть)

(вещественная_часть)

вещественная_часть

- Возвращает `strm`.

- Поведение операторов ввода и вывода более подробно описано на с. 518.

Операторы

`complex operator+ (const complex& cmplx)`

Возвращает `cmplx`.

`complex operator+ (const complex& cmplx)`

- Унарное изменение знака.

- Возвращает `cmplx` с обратными знаками вещественной и мнимой частей.

`complex бинарная_операция (const complex& cmplx1, const complex& cmplx2)`

`complex бинарная_операция (const complex& cmplx1, const T& value)`

`complex бинарная_операция (const T& value, const complex& cmplx2)`

- Все формы возвращают комплексный результат выполнения заданной бинарной операции.

- Параметр `бинарная_операция` — один из следующих операторов:

`operator+`

`operator-`

`operator*`

`operator/`

- Если при вызове оператора передается скалярное значение типа элемента, оно интерпретируется как вещественная часть, а мнимая часть инициализируется значением по умолчанию для своего типа (0 для базовых типов данных).

`bool сравнение (const complex& cmplx1, const complex& cmplx2)`

`bool сравнение (const complex& cmplx1, const T& value)`

`bool сравнение (const T& value, const complex& cmplx1)`

- Возвращает результат сравнения двух комплексных чисел или комплексного числа со скалярным значением.

- Параметр *сравнение* — один из следующих операторов:

```
operator ==
operator !=
```

- Если при вызове оператора передается скалярное значение типа элемента, оно интерпретируется как вещественная часть, а мнимая часть инициализируется значением по умолчанию для своего типа (0 для базовых типов данных).
- Обратите внимание: операторы <, <=, > и >= не поддерживаются.

Трансцендентные функции

```
complex pow (const complex& base, int exp)
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 формы возвращают результат комплексного возведения *base* в степень *exp*, вычисляемый по формуле:

```
exp(exp*log(base))
```

- Разрывы направлены вдоль отрицательной вещественной полуоси.
- Значение, возвращаемое для pow(0,0), определяется реализацией.

```
complex exp (const complex& cplx)
```

Возвращает результат комплексного возведения числа *e* в степень *cplx*.

```
complex sqrt (const complex& cplx)
```

- Возвращает комплексный квадратный корень из *cplx*, находящийся в правой полуплоскости.
- Если аргумент является отрицательным вещественным числом, то возвращаемое значение находится на положительной мнимой полуоси.
- Разрывы направлены вдоль отрицательной вещественной полуоси.

```
complex log (const complex& cplx)
```

- Возвращает комплексный натуральный логарифм *cplx*.
- Если *cplx* является отрицательным вещественным числом, то *imag(log(cplx))*= π .
- Разрывы направлены вдоль отрицательной вещественной полуоси.

```
complex log10 (const complex& cplx)
```

- Возвращает комплексный десятичный логарифм *cplx*.
- Эквивалент:

```
log(cplx)/log(10)
```

- Разрывы направлены вдоль отрицательной вещественной полуоси.

```
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)
```

- Функции возвращают результаты соответствующих комплексных тригонометрических операций с *cmplx*.

Массивы значений

В стандартную библиотеку C++ входит класс *valarray*, предназначенный для обработки массивов числовых значений. Массив значений представляет математическую концепцию линейной последовательности значений. Он имеет всего одно измерение, но специальные правила вычисления индексов и мощные средства выделения подмножеств позволяют имитировать большую размерность. Таким образом, массив значений может использоваться в качестве эффективной базы для операций с векторами и матрицами, а также для обработки математических систем полиномиальных уравнений.

Нетривиальные оптимизации в классах массивов значений обеспечивают хорошее быстродействие при обработке данных. Впрочем, сейчас еще нельзя сказать, насколько важным этот компонент стандартной библиотеки C++ останется в будущем, потому что существуют другие интересные разработки, которые показывают еще лучшие результаты. Один из самых интересных примеров — система *Blitz*. Если вы интересуетесь обработкой числовых данных, обязательно познакомьтесь с ней. За подробностями обращайтесь по адресу <http://monet.uwaterloo.ca/blitz/>.

Качество проектирования классов массивов значений оставляет желать лучшего. В сущности, никто даже не пытался проверить работоспособность итоговой спецификации, поскольку никто не «чувствовал себя ответственным» за эти классы. Люди, включившие массивы значений в стандартную библиотеку C++, покинули комитет по стандартизации задолго до написания стандарта. Например, для работы с массивами значений часто требуются неудобные и долгие преобразования типов (см. с. 532).

Знакомство с массивами значений

Массив значений представляет собой одномерный массив с последовательной нумерацией элементов, начиная с 0. Массивы значений предоставляют средства для выполнения вычислений со всеми элементами (или с их подмножеством) одного или нескольких массивов значений. Например, в следующей команде *a*, *b*, *c*, *x* и *z* могут быть массивами, содержащими сотни числовых значений:

z = *a***x***x* + *b***x* + *c*

К преимуществам такого синтаксиса следует отнести простоту записи. Кроме того, массивы значений обеспечивают хорошую эффективность обработки данных благодаря оптимизациям, предотвращающим создание временных объектов во время выполнения команды. Специальные интерфейсы и вспомогательные классы предоставляют средства для ограничения обработки подмножеством элементов массива значений и для имитации многомерных массивов. В этом отношении массивы значений также упрощают реализацию классов векторов и матриц.

Стандарт гарантирует, что доступ к массивам значений является безальтернативным. Другими словами, обращение к любому значению неконстантного массива значений производится по уникальному пути. Такие гарантии помогают оптимизировать операции с массивами, поскольку компилятору не приходится учитывать возможность обращения к данным по другому пути.

Заголовочный файл

Массивы значений объявляются в заголовочном файле `<valarray>`:

```
#include <valarray>
```

В частности, `<valarray>` содержит объявления следующих классов:

```
namespace std {
    template<class T> class valarray;           // Числовой массив типа T

    class slice;                                // Срез массива значений
    template<class T> class slice_array;

    class gslice;                               // Обобщенный срез
    template<class T> class gslice_array;

    template<class T> class mask_array;

    template<class T> class indirect_array;
}
```

Эти классы имеют следующий смысл.

- `Valarray` — основной класс для работы с массивами числовых значений.
- Классы `slice` и `gslice` предназначены для определения BLAS-совместимых¹ срезов в виде подмножества элементов массива значений.
- Внутренние вспомогательные классы `slice_array`, `gslice_array`, `mask_array` и `indirect_array` предназначены для хранения временных данных. Непосредственное использование этих классов в программном интерфейсе невозможно, они создаются автоматически некоторыми операциями с массивами значений.

Все классы определяются как шаблоны, параметризованные по типу элементов. В принципе элементы могут относиться к произвольному типу данных, но по природе природой массивов значений это должен быть числовой тип.

¹ Библиотека BLAS (Basic Linear Algebra Subprograms) содержит инструментарий для выполнения базовых операций линейной алгебры, включая умножение матриц, решение систем уравнений и простые действия с векторами.

Создание массивов значений

При создании массива значений количество элементов обычно передается в параметре конструктора:

```
std::valarray<int> val(10);      // Массив из десяти нулевых элементов int  
std::valarray<float> val2(5.7,10); // Массив из десяти элементов float.  
                                  // равных 5.7 ( обратите внимание  
                                  // на порядок! )
```

Если конструктор вызывается с одним аргументом, передаваемое значение интерпретируется как размер массива. Элементы инициализируются конструктором по умолчанию для типа элемента; базовые типы инициализируются нулями (инициализация базовых типов конструктором по умолчанию рассматривается на с. 30). Если при вызове конструктора передаются два аргумента, то первый определяет начальное значение, а второй — количество элементов. Такой порядок передачи двух аргументов конструктору отличается от остальных классов стандартной библиотеки C++. Во всех контейнерах STL первый числовый аргумент определяет количество элементов, а второй — их исходное значение.

Массив значений также можно инициализировать обычным массивом:

```
int array[] = { 3, 6, 18, 3, 22 };  
  
// Инициализация массива значений элементами обычного массива:  
std::valarray<int> va3(array,sizeof(array)/sizeof(array[0]));  
  
// Инициализация элементами массива со второго по четвертый  
std::valarray<int> va4(array+1,3);
```

Массив значений создает копии передаваемых данных, поэтому для инициализации могут использоваться временные данные.

Операции с массивами значений

В массивах значений для обращения к элементам определен оператор индексирования `[]`. Как обычно, первому элементу соответствует индекс 0:

```
va[0] = 3 * va[1] + va[2];
```

Также для массивов значений определяются все обычные математические операторы (сложение, вычитание, умножение, деление с остатком, изменение знака, поразрядные операторы, операторы сравнения и логические операторы, а также полный набор операторов присваивания). Эти операторы вызываются для каждого элемента массива значений, обрабатываемого в ходе операции. Таким образом, результат операции с массивом значений представляет собой массив значений, размер которого соответствует размеру operandов; этот массив значений содержит результат поэлементных вычислений. Например, рассмотрим следующую команду:

```
val = va2 * va3;
```

Эта команда эквивалентна последовательности команд:

```
val[0] = va2[0] * va3[0];
val[1] = va2[1] * va3[1];
val[2] = va2[2] * va3[2];
...
```

Если количество элементов в массивах-операндах различно, результат операции не определен.

Конечно, выполнение операций возможно только в том случае, если они поддерживаются типом элементов, а конкретный смысл операции зависит от смысла операции для элементов. Иначе говоря, любая операция с массивом значений сводится к выполнению одинаковых операций с каждым элементом или парой элементов массивов-операндов.

Для бинарных операций один из operandов может быть задан в виде отдельного значения, относящегося к типу элементов. В этом случае значение объединяется с каждым элементом массива значений, переданного во втором операнде. Например, рассмотрим следующую команду:

```
val = 4 * va2;
```

Эта команда эквивалентна последовательности команд:

```
val[0] = 4 * va2[0];
val[1] = 4 * va2[1];
val[2] = 4 * va2[2];
...
```

Помните, что тип отдельного значения должно точно соответствовать типу элементов массива значений. Приведенный выше пример будет работать только в том случае, если элементы массива значений относятся к типу `int`. Например, следующая команда недопустима:

```
std::valarray<double> va(20);
...
va = 4 * va; // ОШИБКА: несоответствие типов
```

Правила бинарных операций также распространяются на операторы сравнения. Оператор `==` не возвращает одну логическую величину — признак равенства массивов. Вместо этого он возвращает новый массив значений того же размера, содержащий элементы типа `bool`; каждый элемент содержит результат сравнения соответствующей пары элементов. Например, рассмотрим следующий фрагмент:

```
std::valarray<double> val(10);
std::valarray<double> va2(10);
std::valarray<double> vab(10);
...
vab = (val == va2);
```

В этом фрагменте последняя команда эквивалентна последовательности команд:

```
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 массивы значений не могут сортироваться оператором `<` и не могут использоваться в качестве элементов контейнеров STL, если проверка их на равенство выполняется оператором `==` (требования к элементам контейнеров STL изложены на с. 143).

Следующая программа представляет собой простой пример использования массивов значений:

```
// num/vall.cpp
#include <iostream>
#include <valarray>
using namespace std;

// Вывод массива значений
template <class T>
void printValarray (const valarray<T>& va)
{
    for (int i=0; i<va.size(); i++) {
        cout << va[i] << ' ';
    }
    cout << endl;
}

int main()
{
    // Определение двух массивов с десятью элементами
    valarray<double> val(10), va2(10);

    // Заполнение первого массива значениями 0.0, 1.1 ... 9.9
    for (int i=0; i<10; i++) {
        val[i] = i * 1.1;
    }

    // Присваивание -1 всем элементам второго массива значений
    va2 = -1;

    // Вывод обоих массивов значений
    printValarray(val);
    printValarray(va2);

    // Вывод минимума, максимума и суммы элементов для первого массива
    cout << "min(): " << val.min() << endl;
```

```

cout << "max(): " << val.max() << endl;
cout << "sum(): " << val.sum() << endl;

// Присваивание содержимого первого массива второму
va2 = val;

// Удаление всех элементов первого массива
val.resize(0);

// Повторный вывод обоих массивов значений
printValarray(val);
printValarray(va2);
}

```

Результат выполнения программы выглядит так:

```

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
min(): 0
max(): 9.9
sum(): 49.5

```

Трансцендентные функции

Трансцендентные функции (тригонометрические и экспоненциальные) определяются по аналогии с математическими операторами. Соответствующая операция также выполняется со всеми элементами массива. Для бинарных операций один операнд может быть отдельным значением, а второй операнд — содержать все элементы массива значений.

Все эти операции определяются в виде глобальных функций (а не в виде функций класса). Это сделано для того, чтобы обеспечить автоматическое преобразование типов подмножеств элементов для обоих операндов (подмножества массивов значений рассматриваются на с. 531).

Второй пример с массивами значений демонстрирует использование трансцендентных операций:

```

// num/val2.cpp
#include <iostream>
#include <valarray>
using namespace std;

// Вывод массива значений
template <class T>
void printValarray (const valarray<T>& va)
{
    for (int i=0; i<va.size(); i++) {
        cout << va[i] << ' ';
    }
    cout << endl;
}

```

```
int main()
{
    // Создание и инициализация массивов значений с девятью элементами
    valarray<double> va(9);
    for (int i=0; i<va.size(); i++) {
        va[i] = i * 1.1;
    }

    // Вывод массива значений
    printValarray(va);

    // Удвоение элементов массива
    va *= 2.0;

    // Повторный вывод массива значений
    printValarray(va);

    // Создание второго массива значений, инициализированного
    // элементами первого массива, увеличенными на 10
    valarray<double> vb(va+10.0);

    // Вывод второго массива значений
    printValarray(vb);

    // Создание третьего массива значений и его инициализация
    // результатом выполнения операций с обоими существующими массивами.
    valarray<double> vc;
    vc = sqrt(va) + vb/2.0 - 1.0;

    // Вывод третьего массива значений
    printValarray(vc);
}
```

Результат выполнения программы выглядит так:

```
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 11.2665 12.8166 14.2332 15.6243 16.9952
```

Подмножества элементов в массивах значений

Оператор индексирования [] перегружается для специальных вспомогательных объектов, различными способами определяющих подмножества элементов. Перегрузка обеспечивает элегантный механизм выполнения операций с частью элементов массива (с доступом как для чтения, так и для записи).

Чтобы определить подмножество массива значений, достаточно указать вместо индекса соответствующее определение, например:

```
va[std::slice(2,4,3)] // Четыре элемента на расстоянии 3.
                      // начиная с индекса 2
va[va>7]           // Все элементы со значением, большим 7
```

Если определение подмножества (такое, как `std::slice(2,4,3)` или `va>7`) используется с константным массивом значений, то выражение возвращает новый массив значений с соответствующими элементами. Но если определение подмножества используется с неконстантным массивом значений, то выражение возвращает временный объект специального вспомогательного класса. Временный объект содержит не данные подмножества, а только его определение. Обработка выражений откладывается до того момента, когда для получения окончательного результата потребуются данные.

Подобный механизм называется *отложенным вычислением*. Отказ от вычисления временных данных экономит время и память. Кроме того, отложенные вычисления обеспечивают ссылочную семантику, то есть подмножество представляет собой логический набор ссылок на исходные данные. Это позволяет использовать его в качестве приемника команды (`l-значения`). Например, подмножеству массива значений можно присвоить результат умножения двух других подмножеств того же массива (примеры приводятся далее).

С другой стороны, отложенные вычисления могут дать неожиданные результаты в ситуациях, когда элементы приемного множества также присутствуют в исходном множестве. По этой причине любые операции с массивами значений гарантированно работают только в том случае, если исходное и приемное множества не имеют общих элементов.

Хорошо продуманное определение подмножеств позволяет наделить массивы значений семантикой двух и более измерений. Это означает, что массивы значений могут использоваться в качестве многомерных массивов.

Существуют четыре варианта определения подмножеств в массивах значений:

- срезы;
- обобщенные срезы;
- отбор по логическому признаку;
- перечисляемые подмножества.

Ниже мы рассмотрим все четыре способа с конкретными примерами.

Проблемы с подмножествами элементов массивов значений

Прежде чем переходить к рассмотрению вариантов определения подмножеств, следует упомянуть одну общую проблему. Работа с подмножествами элементов в массивах значений организована недостаточно хорошо. Вы легко можете создавать подмножества, но при их использовании вместе с другими подмножествами возникают трудности. К сожалению, практически всегда приходится выполнять явное преобразование типа к `valarray`. Дело в том, что в стандартной библиотеке C++ не указано, что подмножества поддерживают те же операции, что и массивы значений.

Предположим, нужно вычислить произведение двух подмножеств и присвоить результат третьему подмножеству. Следующее решение не подходит:

```
// ОШИБКА: отсутствуют преобразования типов
va[std::slice(0,4,3)]
    = va[std::slice(1,4,3)] * va[std::slice(2,4,3)];
```

Вместо этого приходится использовать новые операторы преобразования типа (см. с. 35):

```
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)]);
```

Подойдет также старый механизм преобразования типа:

```
va[std::slice(0,4,3)]
= std::valarray<double>(va[std::slice(1,4,3)]) *
  std::valarray<double>(va[std::slice(2,4,3)]);
```

Ввод таких конструкций — занятие утомительное и чреватое ошибками. Что еще хуже, без хорошей оптимизации работа программы замедлится — при каждом преобразовании создается временный объект, который нужен исключительно для преобразования типа и ни для чего другого.

Следующая шаблонная функция несколько упрощает работу с подмножествами:

```
/* шаблон для преобразования подмножества элементов
   массива значений в массив значений
*/
template <class T>
inline
std::valarray<typename T::value_type> VA(const T& valarray_subset)
{
    return std::valarray<typename T::value_type>(valarray_subset);
```

При использовании этого шаблона предыдущий пример выглядит так:

```
va[std::slice(0,4,3)] = VA(va[std::slice(1,4,3)]) *
                         VA(va[std::slice(2,4,3)]); // OK
```

Однако проблема снижения быстродействия остается.

При работе с элементами заданного типа также можно воспользоваться простым определением типа:

```
typedef valarray<double> VAD;
```

В этом случае запись выглядит так (при условии, что элементы **va** относятся к типу **double**):

```
va[std::slice(0,4,3)] = VAD(va[std::slice(1,4,3)]) *
                         VAD(va[std::slice(2,4,3)]); // OK
```

Срезы

Срез определяется набором индексов, который характеризуется тремя свойствами:

- начальным индексом;
- количеством элементов (размером);
- расстоянием между элементами (шагом).

Порядок передачи этих трех свойств точно соответствует порядку следования параметров конструктора класса `slice`. Например, следующее выражение определяет четыре элемента, начиная с индекса 2, находящихся на расстоянии 3 друг от друга:

```
std::slice(2, 4, 3)
```

Другими словами, выражение определяет такой набор индексов:

```
2 5 8 11
```

Шаг может быть отрицательным. Например, рассмотрим следующее выражение:

```
std::slice(9, 5, -2)
```

Это выражение определяет такой набор индексов:

```
9 7 5 3 1
```

Чтобы определить подмножество элементов массива значений, достаточно передать срез в качестве аргумента оператора индексирования. Например, следующее выражение определяет подмножество массива `va`, содержащее элементы с индексами 2, 5, 8 и 11:

```
va[std::slice(2, 4, 3)]
```

Перед вызовом необходимо проверить правильность всех индексов.

Если подмножество, заданное в виде среза, принадлежит константному массиву значений, то оно образует новый массив значений. Если массив значений не является константным, то подмножество предоставляет ссылочную семантику для работы с исходным массивом значений. Для этой цели определяется вспомогательный класс `slice_array`:

```
namespace std {
    class slice;

    template <class T>
    class slice_array;

    template <class T>
    class valarray {
        public:
            // Срез константного массива значений
            // возвращает новый массив значений
            valarray<T> operator[] (slice) const;

            // Срез неконстантного массива значений возвращает slice_array
            slice_array<T> operator[] (slice);
            ...
    };
}
```

Для объектов `slice_array` определены следующие операции:

- присваивание одного значения всем элементам;
- присваивание другого массива значений (или подмножества массива значений);
- вызовы любых операций вычисляемого присваивания (с такими операторами, как `+=` и `*=`).

Для остальных операций подмножество необходимо преобразовать в массив значений (см. с. 533). Учтите, что класс `slice_array` проектировался исключительно как внутренний вспомогательный класс для работы со срезами, поэтому он должен оставаться невидимым для внешних пользователей. По этой причине все конструкторы и операторы присваивания класса `slice_array<>` объявлены закрытыми.

Например, представленная ниже команда присваивает 2 третьему, шестому, девятому и двенадцатому элементам массива значений `va`:

```
va[std::slice(2,4,3)] = 2;
```

Она эквивалентна следующему набору команд:

```
va[2] = 2;  
va[5] = 2;  
va[8] = 2;  
va[11] = 2;
```

Другая команда возводит в квадрат элементы с индексами 2, 5, 8 и 11:

```
va[std::slice(2,4,3)]  
*= std::valarray<double>(va[std::slice(2,4,3)]);
```

Как упоминалось на с. 533, следующая запись является ошибочной:

```
va[std::slice(2,4,3)] *= va[std::slice(2,4,3)]; // ОШИБКА
```

Но если воспользоваться шаблонной функцией `VA()` (см. с. 533), запись принимает следующий вид:

```
va[std::slice(2,4,3)] *= VA(va[std::slice(2,4,3)]); // OK
```

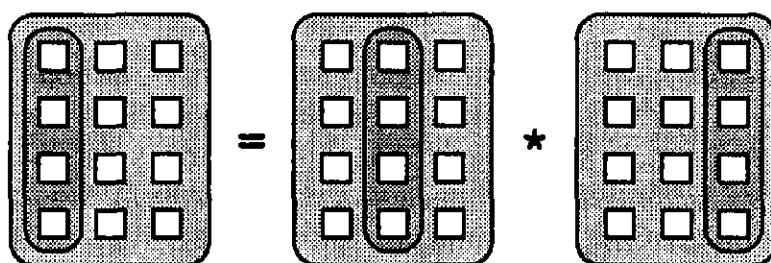
Передавая разные срезы одного массива значений, можно объединить подмножества и сохранить результат в другом подмножестве массива. Например, рассмотрим такую команду:

```
va[std::slice(0,4,3)] = VA(va[std::slice(1,4,3)]) *  
VA(va[std::slice(1,4,3)]);
```

Эта команда эквивалентна следующему набору команд:

```
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 двухмерную матрицу, этот пример представляет собой не что иное, как умножение векторов (рис. 12.1). Однако следует учитывать, что порядок выполнения отдельных присваиваний не определен. Следовательно, если приемное подмножество содержит элементы, используемые в исходном подмножестве, последствия могут быть непредсказуемыми.



$$\text{va[slice(0,4,3)]} = \text{va[slice(1,4,3)]} * \text{va[slice(2,4,3)]}$$

Рис. 12.1. Умножение векторов с использованием срезов

Также возможны и более сложные команды. Например:

```
va[std::slice(0,100,3)]
= std::pow(va[std::slice(1,100,3)]) * 5.0,
VA(va[std::slice(2,100,3)]));
```

Еще раз обратите внимание: отдельное значение (5.0 в данном случае) должно точно соответствовать типу элементов массива значений.

Следующая программа представляет собой полноценный пример использования срезов:

```
// num/slicel.cpp
#include <iostream>
#include <valarray>
using namespace std;

// Вывод массива значений по строкам
template<class 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 с 12 элементами
     * - четыре строки
```

```
* - три столбца
*/
valarray<double> va(12);

// Заполнение массива данными
for (int i=0; i<12; i++) {
    va[i] = i;
}

printValarray (va, 3);

// первый столбец = второй столбец,
// возвещенный в степень третьего столбца
va[slice(0.4.3)] = pow (valarray<double>(va[slice(1.4.3)]),
                           valarray<double>(va[slice(2.4.3)]));

printValarray (va, 3);

// Создание массива значений valarray с троекратным повторением
// третьего элемента va
valarray<double> vb(va[slice(2.4.0)]);

// Умножение третьего столбца на элементы vb
va[slice(2.4.3)] *= vb;

printValarray (va, 3);

// Вывод квадратных корней из элементов второй строки
printValarray (sqrt(valarray<double>(va[slice(3.3,1)])), 3);

// Удвоение элементов третьей строки
va[slice(2.4.3)] = valarray<double>(va[slice(2.4.3)]) * 2.0;

printValarray (va, 3);
}
```

Результат выполнения программы выглядит так:

```
0 1 2
3 4 5
6 7 8
9 10 11
1 1 2
1024 4 5
5.7648y+006 7 8
1e+011 10 11
1 1 4
1024 4 10
5.7648y+006 7 16
1e+011 10 22
32 2 3.16228
```

1 1 8
1024 4 20
5.7648y+006 7 32
1e+011 10 44

Обобщенные срезы

По аналогии со срезами, определяющими подмножество элементов из одного измерения двухмерного массива, обобщенные срезы предназначены для обработки подмножеств элементов многомерных массивов. Вообще говоря, обобщенные срезы характеризуются теми же свойствами, что и обычные срезы:

- начальным индексом;
- количеством элементов (размером);
- расстоянием между элементами (шагом).

Однако в отличие от обычных срезов количество и расстояния между элементами в обобщенных срезах определяются массивами. Количество элементов в таких массивах эквивалентно размерности. Например, обобщенный срез со следующим состоянием эквивалентен обычному срезу, поскольку массив имеет всего одно измерение:

Начальный индекс: 2
Размер: [4]
Шаг: [3]

Иначе говоря, обобщенный срез определяет четыре элемента на расстоянии 3, начинающихся с индекса 2:

2 5 8 11

Однако обобщенный срез с представленным ниже состоянием фактически создает двухмерный массив:

Начальный индекс: 2
Размер: [2 4]
Шаг: [10 3]

Срез определяет две серии из четырех элементов на расстоянии 3, начиная с индекса 2, разделенных расстоянием 10:

2 5 8 11
12 15 18 21

Пример среза с тремя измерениями:

Начальный индекс: 2
Размер: [3 2 4]
Шаг: [30 10 3]

Срез содержит следующие серии элементов:

2 5 8 11
12 15 18 21

```
32 35 38 41  
42 45 48 51
```

```
62 65 68 71  
72 75 78 81
```

Возможность использования массивов для определения размера и шага является единственным отличием между обычными и обобщенными срезами. В остальном обобщенные срезы ведут себя точно так же.

- Чтобы определить подмножество элементов массива значений, достаточно передать обобщенный срез в аргументе оператора [].
- Для константных массивов значений результат представляет собой новый массив значений.
- Для неконстантных массивов значений итоговое выражение представляет собой объект `gslice_array`, представляющий набор элементов массива значений со ссылочной семантикой:

```
namespace std {  
    class gslice;  
  
    template <class T>  
    class gslice_array;  
  
    template <class T>  
    class valarray {  
        public:  
            // Обобщенный срез константного массива значений  
            // возвращает новый массив значений  
            valarray<T> operator[] (const gslice&) const;  
  
            // Обобщенный срез неконстантного массива  
            // значений возвращает gslice_array  
            gslice_array<T> operator[] (const gslice&);  
            ...  
    };  
}
```

- Для класса `gslice_array` определены операторы присваивания (обычные и комбинированные), позволяющие модифицировать элементы подмножества.
- Преобразования типа позволяют объединять обобщенные срезы с другими массивами значений и подмножествами их элементов (см. с. 532).

Следующая программа демонстрирует работу со срезами массивов значений:

```
// num/gslice1.cpp  
#include <iostream>  
#include <valarray>  
using namespace std;  
  
// Построчный вывод трехмерного массива значений  
template<class 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);

    // Заполнение массива значений данными
    for (int i=0; i<24; i++) {
        va[i] = i;
    }

    // Вывод массива значений
    printValarray3D (va, 3, 4);

    // Нам нужны два двумерных подмножества 3x3 элемента
    // в двух 12-элементных массивах
    size_t lengthvalues[] = { 2, 3 };
    size_t stridevalues[] = { 12, 3 };
    valarray<size_t> length(lengthvalues,2);
    valarray<size_t> stride(stridevalues,2);

    // Присвоить второй столбец первых трех строк
    // первому столбцу первых трех строк
    va[gslice(0,length,stride)]
        = valarray<double>(va[gslice(1,length,stride)]);

    // Прибавить и присвоить третью из первых трех строк
    // первой из первых трех строк
    va[gslice(0,length,stride)]
        += valarray<double>(va[gslice(2,length,stride)]);

    // Вывод массива значений
    printValarray3D (va, 3, 4);
}
```

Результат выполнения программы выглядит так:

```
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
```

Определение подмножеств по маске

Следующий способ позволяет определять подмножества элементов по маске — логическому выражению. Например, рассмотрим следующее выражение:

```
va[va > 7]
```

В этом выражении показанное ниже подвыражение возвращает массив размера `va`, в котором для каждого элемента логический признак указывает, превышает ли этот элемент 7:

```
va > 7
```

Оператор индексирования при помощи этого логического массива определяет все элементы, для которых логическое выражение возвращает `true`. Таким образом, следующая конструкция определяет подмножество элементов массива `va`, больших 7.

```
va[va > 7]
```

В остальном маскированные массивы ничем не отличаются от других подмножеств.

- Чтобы определить подмножество элементов массива значений, вы просто передаете массив логических значений в аргументе оператора [] массива значений.
- Если массив значений является константным, то полученное выражение определяет новый массив значений.
- Для неконстантных массивов значений полученное выражение определяет объект `mask_array`, представляющий набор элементов массива значений со ссылочной семантикой:

```
namespace std {  
    template <class T>  
    class mask_array:
```

```

template <class T>
class valarray {
public:
    // Маскирование константного массива значений
    // возвращает новый массив значений
    valarray<T> operator[] (const gslice&) const;

    // Маскирование неконстантного массива
    // значений возвращает mask_array
    mask_array<T> operator[] (const valarray<bool>&);

    ...
};

}

```

- Для класса `mask_array` определены операторы присваивания (обычные и комбинированные), позволяющие модифицировать элементы подмножества.
- Преобразования типа позволяют объединять маскированные массивы с другими массивами значений и подмножествами их элементов (см. с. 532).

Следующая программа показывает, как использовать маскированные подмножества элементов:

```

// num/masked1.cpp
#include <iostream>
#include <valarray>
using namespace std;

// Построчный вывод массива значений
template<class 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 с 12 элементами
     * - четыре строки
     * - три столбца
     */
    valarray<double> va(12);

    // Заполнение массива значений
    for (int i=0; i<12; i++) {
        va[i] = i;
    }
}

```

```

printValarray (va, 3);

// Присваивание 77 всем элементам, меньшим 5
va[va<5.0] = 77.0;

// Присваивание 100 всем значениям, большим 5, но меньшим 9
va[v>5.0 && v<9.0]
    = valarray<double>(va[v>5.0 && v<9.0]) + 100.0;

printValarray (va, 3);

```

Результат выполнения программы выглядит так:

```

) 1 2
) 4 5
) 7 8
) 10 11

) 7 77 77
) 77 77 5
) 106 107 108
) 10 11

```

Обратите внимание: тип числового значения, сравниваемого с массивом значений, должен точно соответствовать типу массива. Следовательно, попытка компиляции программы, в которой при сравнении с массивом значений `double` используются значения `int`, завершится неудачей:

```

valarray<double> va(12);
...
va[v<5] = 77; // ОШИБКА

```

Перечисляемые подмножества

Четвертый и последний вариант определения подмножеств элементов — перечисляемые подмножества. Подмножество элементов массива значений определяется простой передачей массива с индексами. При этом индексы не обязаны следовать в порядке сортировки и могут включаться в массив дважды.

В остальном перечисляемые подмножества ничем не отличаются от других подмножеств массивов значений.

- С Чтобы определить подмножество элементов массива значений, вы просто передаете массив с элементами типа `size_t` в аргументе оператора `[]` массива значений.
- С Если массив значений является константным, то полученное выражение определяет новый массив значений.
- С Для неконстантных массивов значений полученное выражение определяет объект `indirect_array`, представляющий набор элементов массива значений со ссылочной семантикой:

```

namespace std {
    template <class T>
    class indirect_array:

```

```

template <class T>
class valarray {
public:
    // Индексирование константного массива значений
    // возвращает новый массив значений
    valarray<T> operator[] (const valarray<size_t>&) const;

    // Индексирование неконстантного массива
    // значений возвращает indirect_array
    indirect_array<T> operator[] (const valarray<size_t>&);

    ...
};

}

```

- Для класса `indirect_array` определены операторы присваивания (обычные и комбинированные), позволяющие модифицировать элементы подмножества.
- Преобразования типа позволяют объединять перечисляемые подмножества с другими массивами значений и подмножествами их элементов (см. с. 532).

Следующая программа показывает, как использовать перечисляемые подмножества элементов:

```

// num/ind1.cpp
#include <iostream>
#include <valarray>
using namespace std;

// Построчный вывод массива значений
template<class 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()
{
    // Создание массива значений с 12 элементами
    valarray<double> va(12);

    // Инициализация массива значениями 1.01, 2.02, ... 12.12
    for (int i=0; i<12; i++) {
        va[i] = (i+1) * 1.01;
    }
    printValarray(va,4);

    /* Создание массива индексов
     * - ВНИМАНИЕ: элементы должны относиться к типу size_t
    */
}

```

```
/*
valarray<size_t> idx(4);
idx[0] = 8;
idx[1] = 0;
idx[2] = 3;
idx[3] = 7;

// Вывод девятого, первого, четвертого и восьмого элементов
// с использованием перечисляемого подмножества
printValarray(valarray<double>(va[idx]), 4);

// Изменение первого и четвертого элементов
// и повторный вывод перечисляемого подмножества
va[0] = 11.11;
va[3] = 44.44;
printValarray(valarray<double>(va[idx]), 4);

// Выбор второго, третьего, шестого и девятого элементов
// и присваивание им 99
idx[0] = 1;
idx[1] = 2;
idx[2] = 5;
idx[3] = 8;
va[idx] = 99;

// Вывод всего массива значений
printValarray (va, 4);
}
```

Переменная `idx` используется для определения подмножеств элементов массива значений `va`. Результат выполнения программы выглядит так:

```
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
```

Строение класса `valarray`

В поддержке массивов значений центральное место занимает класс `valarray<>`. Он определяется как шаблонный класс, параметризованный по типу элементов:

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

Размер не является частью типа. Это означает, что теоретически вы можете обрабатывать массивы значений разных размеров и изменять размеры. Тем не менее изменение размера поддерживается только для двухшаговой инициализации (создание и изменение размера), неизбежной при работе с массивами значений. Учтите, что результат объединения массивов разных размеров не определен.

Операции создания, копирования и удаления

`valarray::valarray ()`

- Конструктор по умолчанию.
- Создает пустой массив значений.
- Конструктор определен только для создания массивов, элементами которых являются массивы значений. Следующим шагом должно быть назначение правильного размера функцией `resize()`.

`explicit valarray::valarray (size_t num)`

- Создает массив значений, содержащий *num* элементов.
- Элементы инициализируются конструктором по умолчанию (0 для базовых типов данных).

`valarray::valarray (const T& value, size_t num)`

- Создает массив значений, содержащий *num* элементов.
- Элементы инициализируются значением *value*.
- Обратите внимание на нестандартный порядок следования параметров. В интерфейсе остальных классов стандартной библиотеки C++ параметр *num* идет первым, а параметр *value* — вторым.

`valarray::valarray (const T* array, size_t num)`

- Создает массив значений, содержащий *num* элементов.
- Элементы инициализируются значениями элементов массива *array*.
- Перед вызовом необходимо убедиться в том, что массив *array* содержит не менее *num* элементов; в противном случае вызов приводит к непредсказуемым последствиям.

`valarray::valarray (const valarray& va)`

- Копирующий конструктор.
- Создает массив значений как копию *va*.

`valarray::~valarray ()`

- Деструктор.
- Уничтожает все элементы и освобождает память.

В добавок массивы значений могут создаваться объектами внутренних вспомогательных классов `slice_array`, `gslice_array`, `mask_array` и `indirect_array`. За подробностями обращайтесь соответственно на с. 551, 553, 554 и 555.

Операции присваивания

`valarray& valarray::operator= (const valarray& va)`

- Присваивает элементы массива значений *va*.
- Если массив *va* имеет другой размер, присваивание приводит к непредсказуемым последствиям.
- Значение элемента в левой части любой операции присваивания массиву значений не должно зависеть от значений других элементов, находящихся в левой части. Другими словами, если операция присваивания перезаписывает значения, используемые в правой части операции присваивания, это приводит к непредсказуемым последствиям. Следовательно, элементы из левой части вообще не могут указываться в выражении, находящемся в правой части. Это ограничение объясняется тем, что порядок вычисления результата при обработке массивов значений не определен. Дополнительная информация приведена на с. 532 и 536.

`valarray& valarray::operator= (const T& value)`

- Присваивает *value* каждому элементу массива значений¹.
- Размер массива значений не изменяется, указатели и ссылки на элементы остаются действительными.

В добавок допускается присваивание значений внутренних вспомогательных классов `slice_array`, `gslice_array`, `mask_array` и `indirect_array`. За подробностями обращайтесь соответственно на с. 551, 553, 554 и 555.

Функции класса

Класс `valarray` определяет следующие функции.

`size_t valarray::size () const`

Возвращает текущее количество элементов².

`void valarray::resize (size_t num)`
`void valarray::resize (size_t num, T value)`

- Обе формы изменяют размер массива значений до величины *num*.
- При увеличении размера новые элементы инициализируются конструктором по умолчанию или значением *value* соответственно.
- Обе формы делают недействительными все указатели и ссылки на элементы массива значений.
- Функции поддерживаются только для создания массивов, элементами которых являются массивы значений. После создания массива значений конструктором по умолчанию следует задать его правильный размер вызовом `resize()`.

¹ В предыдущих версиях STL эта операция выполнялась функцией `fill()`.

² В предыдущих версиях STL функция `size()` называлась `length()`.

`T valarray::min () const`

`T valarray::max () const`

- Первая форма возвращает минимальное значение среди элементов массива значений.
- Вторая форма возвращает максимальное значение среди элементов массива значений.
- Элементы сравниваются операторами `<` и `>`, поэтому эти операторы должны быть определены для типа элементов.
- Если массив значений не содержит элементов, то возвращаемые значения не определены.

`T valarray::sum () const`

- Возвращает сумму всех элементов.
- Элементы обрабатываются оператором `+=`, поэтому этот оператор должен быть определен для типа элементов.
- Если массив значений не содержит ни одного элемента, 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 содержит то же количество элементов.
- Элементы в позициях, освободившихся в результате сдвига, инициализируются конструктором по умолчанию.
- Направление сдвига зависит от знака *num*:
 - если значение *num* положительно, элементы сдвигаются влево/вперед (с уменьшением индекса);
 - если значение *num* отрицательно, элементы сдвигаются вправо/назад (с увеличением индекса);

`valarray valarray::cshift (int num) const`

- Возвращает новое значение, в котором все элементы циклически сдвинуты на *num* позиций.
- Возвращаемый массив значений содержит то же количество элементов.
- Направление сдвига зависит от знака *num*:
 - если значение *num* положительно, элементы сдвигаются влево/вперед (с уменьшением индекса или вставкой элемента в конце);
 - если значение *num* отрицательно, элементы сдвигаются вправо/назад (с увеличением индекса или вставкой элемента в начале).

```
valarray valarray::apply (T op(T)) const
valarray valarray::apply (T op(const T&)) const
```

- Обе формы возвращают новый массив значений с элементами, обработанными предикатом *op()*.
- Возвращаемый массив значений содержит то же количество элементов.
- Для каждого элемента **this* вызывается предикат *op(elem)*, а соответствующий элемент возвращаемого массива инициализируется результатом вызова.

Обращение к элементам

```
T& valarray::operator[] (size_t idx)
T valarray::operator[] (size_t idx) const
```

- Обе формы возвращают элемент массива значений с индексом *idx* (первый элемент с индексом 0).
- Неконстантная версия возвращает ссылку. Это означает, что элемент, возвращаемый оператором, может использоваться для модификации массива значений. Ссылка гарантированно остается действительной все время, пока существует массив значений и для него не вызываются функции, изменяющие размер массива.

Операторы массивов значений

Унарные операторы массивов значений имеют следующий формат:

```
valarray valarray::унарный_оператор () const
```

- Унарный оператор возвращает новый массив значений со всеми элементами **this*, модифицированными оператором.
- Параметр *унарный_оператор* – один из следующих операторов:

```
operator +
operator -
operator ~
operator !
```

- Оператор *!* возвращает тип *valarray<bool>*.

Бинарные операторы для массивов значений (кроме операторов сравнения и присваивания) имеют следующий формат:

```
valarray бинарный_оператор (const valarray& va1, const valarray& va2)
valarray бинарный_оператор (const valarray& va, const T& value)
valarray бинарный_оператор (const T& value, const valarray& va)
```

- Операторы возвращают новый массив значений, размер которого соответствует размеру *va*, *va1* или *va2*. Новый массив значений содержит результат применения бинарного оператора к каждой паре значений.

- Если при вызове оператора передается только отдельное значение *value*, оно комбинируется с каждым элементом *va*.
- Параметр *бинарный_оператор* — один из следующих операторов:

```
operator +
operator -
operator *
operator /
operator %
operator ^
operator &
operator |
operator <<
operator >>
```

- Если *va1* и *va2* содержат разное количество элементов, результат не определен.

Логические операторы и операторы сравнения работают по той же схеме, но они возвращают массив значений с логическими элементами:

```
valarray<bool> логический_оператор (const valarray& va1,
                                         const valarray& va2)
valarray<bool> логический_оператор (const valarray& va1,
                                         const T& value)
valarray<bool> логический_оператор (const T& value,
                                         const valarray& va)
```

- Операторы возвращают новый массив значений, размер которого соответствует размеру *va*, *va1* или *va2*. Новый массив значений содержит результат применения логического оператора к каждой паре значений.
- Если при вызове оператора передается только отдельное значение *value*, оно комбинируется с каждым элементом *va*.
- Параметр *логический_оператор* — один из следующих операторов:

```
operator ==
operator !=
operator <
operator <=
operator >
operator >=
operator &&
operator ||
```

- Если *va1* и *va2* содержат разное количество элементов, результат операции не определен.
- Ссылки и указатели на модифицированные элементы остаются действительными все время, пока существует массив значений и для него не вызываются функции, изменяющие размер массива.

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& va1, const valarray& va2)
valarray atan2 (const valarray& va, const T& value)
valarray atan2 (const T& value, const valarray& va)
```

- Все перечисленные функции возвращают новый массив значений, размер которого соответствует размеру *va*, *va1* или *va2*. Новый массив значений содержит результат вызова функции для каждой пары значений.
- Если *va1* и *va2* содержат разное количество элементов, результат операции не определен.

Классы подмножеств элементов

Здесь подробно описываются классы подмножеств элементов массивов значений. Но поскольку эти классы очень просты и поддерживают небольшое количество операций, будут приведены только их объявления с комментариями.

Классы `slice` и `slice_array`

Объекты класса `slice_array` создаются при передаче среза `slice` в индексе неконстантного массива значений:

```
namespace std {
    template<class T>
    class valarray {
        public:
            ...
            slice_array<T> operator[](slice);
            ...
    };
}
```

Ниже приведено точное определение открытого интерфейса класса `slice`:

```
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. Функции `start()`, `size()` и `stride()` предназначены для получения свойств среза.

Класс `slice_array` поддерживает следующие операции:

```
namespace std {
    template <class 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&);

            ...
    };
}
```

Учтите, что класс `slice_array` проектировался исключительно как внутренний вспомогательный класс для работы со срезами, который должен оставаться невидимым для внешних пользователей. По этой причине все конструкторы и операторы присваивания класса `slice_array<T>` объявлены закрытыми.

Классы `gslice` и `gslice_array`

Объекты класса `gslice_array` создаются при передаче среза `gslice` в индексе неконстантного массива значений:

```
namespace std {
    template<class T>
    class valarray {
        public:
            ...
            gslice_array<T> operator[](const gslice&);
            ...
    };
}
```

Ниже приведено точное определение открытого интерфейса класса `gslice`:

```
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 создаст пустое подмножество. Функции `start()`, `size()` и `stride()` предназначены для получения свойств обобщенного среза.

Класс `gslice_array` поддерживает следующие операции:

```
namespace std {
    template <class 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>&);

    ...
};

}

```

По аналогии с классом `slice_array` класс `gslice_array` проектировался исключительно как внутренний вспомогательный класс для работы с обобщенными срезами, который должен оставаться невидимым для внешних пользователей. По этой причине все конструкторы и операторы присваивания класса `gslice_array<T>` объявлены закрытыми.

Класс `mask_array`

Объекты класса `mask_array` создаются при передаче `valarray<bool>` в индексе не-константного массива значений:

```

namespace std {
    template<class T>
    class valarray {
        public:
            ...
            mask_array<T> operator[](const valarray<bool>&);

            ...
    };
}

```

Класс `mask_array` поддерживает следующие операции:

```

namespace std {
    template <class 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>&);  
    ...  
};  
}
```

Класс `mask_array` также проектировался исключительно как внутренний вспомогательный класс для работы с маскированными подмножествами, который должен оставаться невидимым для внешних пользователей. По этой причине все конструкторы и операторы присваивания класса `mask_array<T>` объявлены закрытыми.

Класс `indirect_array`

Объекты класса `indirect_array` создаются при передаче `valarray<size_t>` в индексе неконстантного массива значений:

```
namespace std {  
    template<class T>  
    class valarray {  
        public:  
            ...  
            indirect_array<T> operator[](const valarray<size_t>&);  
            ...  
    };  
}
```

Класс `indirect_array` поддерживает следующие операции:

```
namespace std {  
    template <class 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>&);

}:
}

```

Класс `indirect_array` также проектировался исключительно как внутренний вспомогательный класс, который должен оставаться невидимым для внешних пользователей. По этой причине все конструкторы и операторы присваивания класса `indirect_array<T>` объявлены закрытыми.

Глобальные математические функции

В заголовочных файлах `<cmath>` и `<cstdlib>` определяются глобальные математические функции, унаследованные из языка С. Эти функции перечислены в табл. 12.8 и 12.9¹.

Таблица 12.8. Функции, определяемые в заголовочном файле `<cmath>`

Функция	Описание
<code>pow()</code>	Возведение в степень
<code>exp()</code>	Экспонента
<code>sqrt()</code>	Квадратный корень
<code>log()</code>	Натуральный логарифм
<code>log10()</code>	Десятичный логарифм
<code>sin()</code>	Синус
<code>cos()</code>	Косинус
<code>tan()</code>	Тангенс
<code>sinh()</code>	Гиперболический синус
<code>cosh()</code>	Гиперболический косинус
<code>tanh()</code>	Гиперболический тангенс
<code>asin()</code>	Арксинус
<code>acos()</code>	Арккосинус
<code>atan()</code>	Арктангенс
<code>atan2()</code>	Арктангенс частного
<code>ceil()</code>	Округление вещественного числа вверх до ближайшего целого
<code>floor()</code>	Округление вещественного числа вниз до ближайшего целого
<code>fabs()</code>	Модуль (абсолютное значение) для типа <code>float</code>
<code>fmod()</code>	Остаток после деления

¹ По историческим причинам некоторые числовые функции определяются не в `<cmath>`, а в файле `<cstdlib>`.

Функция	Описание
frexp()	Преобразование вещественного числа в целую и дробную части
ldexp()	Умножение вещественного числа на целую степень 2
modf()	Извлечение знаковой целой и дробной частей из вещественного числа

Таблица 12.9. Функции, определяемые в заголовочном файле <cstdlib>

Функция	Описание
abs()	Модуль (абсолютное значение) для типа int
labs()	Модуль (абсолютное значение) для типа long
div()	Частное и остаток от деления для типа int
ldiv()	Частное и остаток от деления для типа long
rand()	Инициализация генератора случайных чисел
rand()	Получение следующего случайного числа

В языке C++, в отличие от C, операции могут перегружаться для разных типов, из-за чего некоторые числовые функции С становятся лишними. Например, в С определены функции `abs()`, `labs()` и `fabs()` для вычисления модуля типов `int`, `long` и `double` соответственно. В C++ функция `abs()` перегружается для различных типов данных, что позволяет использовать ее со всеми типами.

В частности, все математические функции перегружены для вещественных типов `float`, `double` и `long double`. Однако при этом возникает важный побочный эффект: при передаче целого значения выражение становится неоднозначным¹:

```
std::sqrt(7)           // НЕОДНОЗНАЧНОСТЬ: sqrt(float), sqrt(double)
                      // или sqrt(long double)?
```

Вместо этого приходится использовать такую запись:

```
std::sqrt(7.0)         // OK
```

А при использовании переменной нужно писать так:

```
int x;
...
std::sort(float(x))   // OK
```

Разработчики библиотек решают эту проблему по-разному: одни не включают перегруженные версии, другие следуют стандарту (перегрузка для всех вещественных типов), третьи предлагают перегрузку для всех числовых типов, а некоторые позволяют выбрать нужный вариант на уровне препроцессора. Это означает, что на практике нельзя быть уверенным как в наличии неоднозначности, так и в ее отсутствии. Чтобы обеспечить переносимость кода, всегда следите за точным соответствием типов аргументов.

¹ Спасибо Дэвиду Вандеворду (David Vandevoorde) за это пояснение.

13 Ввод-вывод с использованием потоковых классов

Классы ввода-вывода являются важнейшими классами стандартной библиотеки C++, — программа, которая не вводит и не выводит данные, вряд ли принесет много пользы. Более того, классы ввода-вывода стандартной библиотеки C++ не ограничиваются операциями с файлами, экраном и клавиатурой. Они создают расширяемую архитектуру для форматирования произвольных данных и работы с произвольными «внешними представлениями».

Библиотека *IOStream* (как называется совокупность классов ввода-вывода) — единственная часть стандартной библиотеки C++, которая широко использовалась до стандартизации C++. В ранние поставки систем C++ включались классы, разработанные в AT&T и ставшие фактическим стандартом ввода-вывода. Хотя в дальнейшем эти классы адаптировались для интеграции со стандартной библиотекой C++ и выполнения некоторых новых функций, базовые принципы, заложенные в классы библиотеки *IOStream*, остались неизменными.

Эта глава начинается с общего обзора важнейших компонентов и принципов потокового ввода-вывода, после чего подробно рассматриваются возможности практического применения библиотеки *IOStream* — от простого форматирования до интеграции с новыми внешними представлениями (тема, которая часто понимается неправильно).

Чтобы подробно рассмотреть все аспекты, относящиеся к библиотеке *IOStream*, понадобилась бы отдельная книга. За дополнительной информацией обращайтесь к книгам, посвященным потоковому вводу-выводу, или справочникам по стандартной библиотеке C++.

Большое спасибо Дитмару Кюлю (Dietmar Kühl), эксперту по вопросам ввода-вывода и интернационализации в стандартной библиотеке C++, который помогал мне в работе над этой главой и написал часть материала.

Для читателей, знакомых со «старой» библиотекой *IOStream*, перечислим изменения, внесенные в процессе стандартизации. Хотя основные принципы потоковых классов ввода-вывода остались неизменными, были добавлены некоторые важные новшества, расширяющие возможности настройки и адаптации.

- Была проделана работа по интернационализации ввода-вывода.
- Потоковые классы для символьных массивов типа `char*` были заменены классами, использующими строковые типы стандартной библиотеки C++. Старые

классы по-прежнему поддерживаются для обеспечения совместимости, но они считаются устаревшими¹.

- Обработка ошибок была интегрирована с обработкой исключений.
- Библиотечные классы `IOStream` с поддержкой присваивания (имена таких классов заканчиваются суффиксом `_withassign`) были заменены новыми средствами, доступными для всех потоковых классов.
- Классы библиотеки `IOStream` преобразованы в шаблоны, что позволяет поддерживать разные представления символов. Однако в результате возникает побочный эффект — нельзя использовать простые опережающие объявления потоковых классов:

```
class ostream; // Ошибка
```

Соответствующие объявления были собраны в специальном заголовочном файле, поэтому вместо них следует использовать новый заголовок:

```
#include <iostream> // OK
```

- Все символические имена библиотеки `IOStream`, как и в остальных компонентах стандартной библиотеки C++, объявляются в пространстве имен `std`.

Общие сведения о потоках ввода-вывода

Прежде чем переходить к подробному описанию потоковых классов, мы кратко рассмотрим общеизвестные аспекты работы с потоками данных — это заложит основу для дальнейшего изложения материала. Читатели, хорошо знакомые с потоковым вводом-выводом, могут пропустить этот раздел.

Потоковые объекты

В C++ операции ввода-вывода выполняются при помощи потоков данных. Согласно принципам объектно-ориентированного программирования, поток данных представляет собой объект, свойства которого определяются классом. Вывод интерпретируется как запись данных в поток, а ввод — как чтение данных из потока. Для стандартных каналов ввода-вывода существуют стандартные глобальные объекты.

Потоковые классы

Специализированные разновидности ввода-вывода (ввод, вывод, операции с файлами) представлены в библиотеке разными классами. Среди потоковых классов центральное место занимают следующие:

- класс `istream` — входной поток, используемый для чтения данных;
- класс `ostream` — выходной поток, используемый для записи данных.

¹ Термин «устаревший» (*deprecated*) означает, что использовать то или иное средство не рекомендуется, поскольку существует новая, более совершенная замена. Кроме того, устаревшие средства с большой вероятностью будут исключены из будущих версий стандарта.

Оба класса представляют собой специализации шаблонов `basic_istream<>` и `basic_ostrream<>` для типа символов `char`. Библиотека IOStream не зависит от конкретного типа символов — для большинства классов библиотеки этот тип передается в аргументе шаблона. Параметризация по типу символов является аналогом параметризации строковых классов и используется при интернационализации программ (см. главу 14).

Здесь мы ограничимся рассмотрением ввода и вывода в «узких» потоках данных, то есть в потоках данных с типом символов `char`. Потоки данных с другими типами символов будут рассмотрены во второй половине главы.

Глобальные потоковые объекты

В библиотеке IOStream определено несколько глобальных объектов типов `istream` и `ostream`. Эти объекты соответствуют стандартным каналам ввода-вывода.

- Объект `cin` (класс `istream`) представляет стандартный входной канал, используемый для ввода пользовательских данных. Он соответствует потоку данных `stdin` в языке С. Обычно операционная система связывает этот канал с клавиатурой.
- Объект `cout` (класс `ostream`) представляет стандартный выходной канал, предназначенный для вывода результатов работы программы. Он соответствует потоку данных `stdout` в языке С. Обычно операционная система связывает этот канал с монитором.
- Объект `cerr` (класс `ostream`) представляет стандартный канал, предназначенный для вывода всевозможных сообщений об ошибках. Он соответствует потоку данных `stderr` в языке С. Обычно операционная система также связывает этот канал с монитором. По умолчанию вывод в `cerr` не буферизуется.
- Объект `clog` (класс `ostream`) представляет стандартный канал для регистрации данных и не имеет аналогов в языке С. По умолчанию этот поток данных связывается с тем же приемником, что и `cerr`, но вывод в `clog` буферизуется.

Отделение «нормального» вывода от сообщений об ошибках позволяет по-разному обойтись с этими двумя категориями потоков данных при выполнении программы. Например, нормальный вывод программы можно перенаправить в файл, тогда как сообщения об ошибках будут выводиться на консоль. Конечно, для этого операционная система должна поддерживать перенаправление стандартных каналов ввода-вывода (впрочем, в большинстве операционных систем такая возможность существует). Разделение стандартных каналов берет свое начало в механизме перенаправления ввода-вывода системы UNIX.

Потоковые операторы

Операторы сдвига `>>` и `<<` были перегружены для потоковых классов и означают соответственно ввод и вывод. При помощи этих операторов можно выполнять каскадные операции ввода-вывода.

Например, следующий цикл при каждой итерации читает из стандартного входного потока данных два целых числа (пока вводятся только целые числа) и записывает их в стандартный выходной поток данных:

```
int a, b;
// Пока операции ввода a и b проходят успешно
while (std::cin >> a >> b {
    // Вывод a и b
    std::cout << "a: " << a << "b: " << b << std::endl;
}
```

Манипуляторы

В конце большинства команд потокового ввода-вывода записывается так называемый манипулятор:

```
std::cout << std::endl
```

Манипуляторы — специальные объекты, предназначенные для управления потоком данных. Часто манипуляторы изменяют только режим интерпретации ввода или форматирования вывода (например, манипуляторы выбора системы счисления `dec`, `hex` и `oct`). Это означает, что манипуляторы потока данных `ostream` не всегда создают выходные данные, а манипуляторы потока данных `istream` не всегда интерпретируют ввод. Однако некоторые манипуляторы выполняют непосредственные действия — очистку выходного буфера, переключение в режим игнорирования пропусков при вводе и т. д.

Манипулятор `endl` обозначает «конец строки», а при его выводе выполняются две операции.

1. Отправка признака новой строки (то есть символа `\n`) в выходной поток данных.
2. Очистка выходного буфера (принудительный вывод всех буферизованных данных методом `flush()`).

Наиболее важные манипуляторы библиотеки `IOStream` перечислены в табл. 13.1. Более подробное описание манипуляторов (в том числе и определенных в библиотеке `IOStream`) приведено на с. 586. Также там рассказано, как определить пользовательский манипулятор.

Таблица 13.1. Важнейшие манипуляторы библиотеки `IOStream`

Манипулятор	Класс	Описание
<code>endl</code>	<code>ostream</code>	Вывод <code>\n</code> и очистка выходного буфера
<code>ends</code>	<code>ostream</code>	Вывод <code>\0</code>
<code>flush</code>	<code>ostream</code>	Очистка выходного буфера
<code>ws</code>	<code>istream</code>	Чтение с игнорированием пропусков

Простой пример

Следующий пример демонстрирует работу с потоковыми классами. Программа читает два вещественных числа и выводит их произведение.

```
// io/iol.cpp
#include <cstdlib>
#include <iostream>
using namespace std;

int main()
{
    double x, y;           // Операнды

    // Вывод заголовка
    cout << "Multiplication of two floating point values" << endl;

    // Чтение первого операнда
    cout << "first operand: ";
    if (!(cin >> x)) {
        /* Ошибка ввода
         * => вывести сообщение и завершить программу с кодом ошибки
         */
        cerr << "error while reading the first floating value"
            << endl;
        return EXIT_FAILURE;
    }

    // Чтение второго операнда
    cout << "second operand: ";
    if (!(cin >> y)) {
        /* Ошибка ввода
         * => вывести сообщение и завершить программу с кодом ошибки
         */
        cerr << "error while reading the second floating value"
            << endl;
        return EXIT_FAILURE;
    }

    // Вывод operandов и результата
    cout << x << " times " << y << " equals " << x * y << endl;
}
```

Основные потоковые классы и объекты

Иерархия потоковых классов

Потоковые классы библиотеки `iostream` образуют иерархию, изображенную на рис. 13.1. Для шаблонных классов в верхней строке указано имя шаблона, а в нижней — имена специализаций для типов `char` и `wchar_t`.

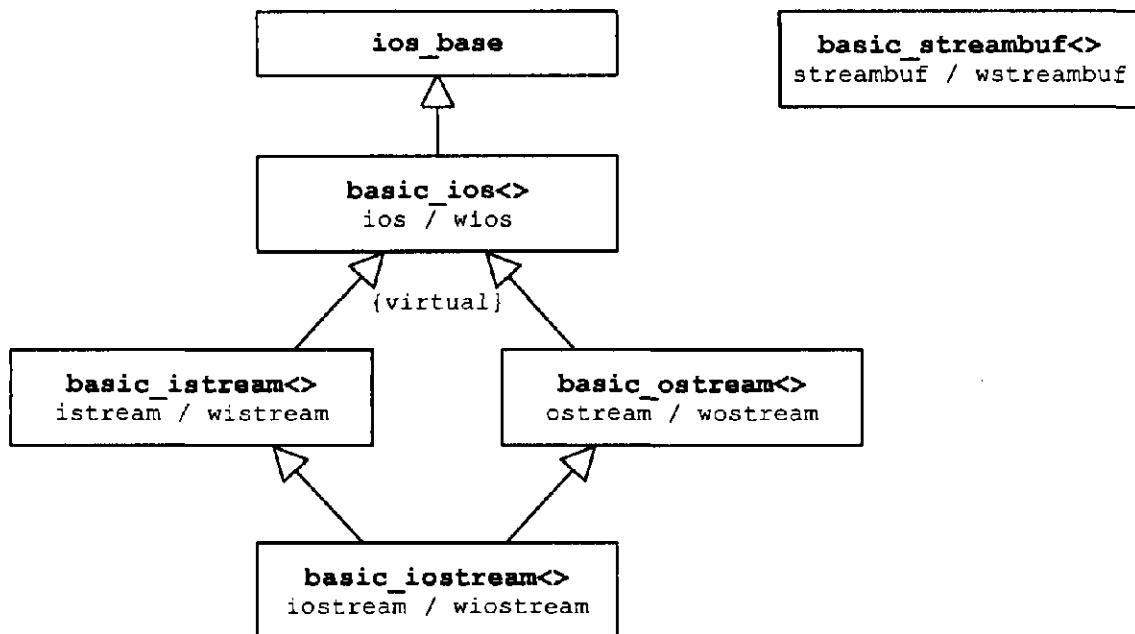


Рис. 13.1. Иерархия основных потоковых классов

Ниже перечислены задачи, решаемые классами этой иерархии.

- Базовый класс `ios_base` определяет свойства всех потоковых классов, не зависящие от типа и трактовок символов. Класс в основном состоит из компонентов и функций, предназначенных для управления состоянием и флагами формата.
- Шаблон класса `basic_ios<>`, производный от `ios_base<>`, определяет общие свойства всех потоковых классов, зависящие от типа и трактовок символов. В число этих свойств входит определение буфера, используемого потоком данных. Буфер представлен объектом класса, производным от базового класса `basic_streambuf<>`, с соответствующей специализацией. Фактически именно он выполняет операции чтения/записи.
- Шаблоны `basic_istream<>` и `basic_oiostream<>`, виртуально производные от `basic_ios<>`, определяют объекты, которые могут использоваться соответственно для чтения и записи. Эти классы, как и `basic_ios<>`, оформлены в виде шаблонов, параметризованных по типу и трактовкам символов. Если проблемы интернационализации несущественны, задействуются специализации этих классов для типа символов `char` (а именно `istream` и `ostream`).
- Шаблон `basic_iostream<>` является производным от двух шаблонов — `basic_istream<>` и `basic_oiostream<>`. Он определяет объекты, которые могут использоваться как для чтения, так и для записи.
- Шаблон `basic_streambuf<>` занимает центральное место в библиотеке IOStream. Он определяет интерфейс всех представлений, записываемых в потоки данных или читаемых из потоков данных, и используется другими потоковыми классами для фактического чтения или записи символов. Для получения доступа к некоторым внешним представлениям классы объявляются производными от `basic_streambuf<>`. Подробности приведены далее.

Назначение потоковых буферных классов

Библиотека `IOStream` проектировалась со строгим разделением обязанностей. Классы, производные от `basic_ios`, «всего лишь» ограничиваются форматированием данных¹. Операции чтения и записи символов выполняются потоковыми буферами, которые представлены объектами, подчиненными по отношению к классу `basic_ios`. Потоковые буфера обеспечивают выполнение чтения/записи в символьных буферах и помогают абстрагироваться от внешнего представления (например, файлов или строковых данных).

Потоковые буфера играют важную роль при выполнении ввода-вывода с новыми внешними представлениями (например, сокетами или компонентами графического интерфейса), перенаправлении потоков данных или их конвейерном объединении (например, при сжатии выходных данных перед их передачей в другой поток данных). Кроме того, потоковые буфера обеспечивают синхронизацию при одновременном вводе-выводе с одним внешним представлением. Дополнительная информация приведена на с. 613.

Потоковые буфера упрощают определение новых «внешних представлений» (скажем, предназначенных для работы с новым носителем данных). Для этого требуется лишь объявить новый потоковый буферный класс, производный от `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. На с. 636 рассказано, как определяются новые потоковые буфера для работы со специальными носителями данных.

Подробные определения классов

Как и все шаблонные классы библиотеки `IOStream`, шаблон `basic_ios<>` параметризуется по двум аргументам, а его определение выглядит так:

```
namespace std {
    template <class charT,
              class traits = char_traits<charT> >
    class basic_ios:
}
```

В аргументах шаблона передается тип символов, используемый потоковыми классами, и класс с описанием трактовок этого типа.

В частности, трактовки класса определяют признак конца файла² и способы копирования/перемещения ряда символов. Как правило, тип символов ассоциируется с определенным набором трактовок, поэтому будет вполне логично определить шаблонный класс, специализируемый для конкретных типов символов.

¹ На самом деле они не делают даже этого! Форматирование поручается соответствующим фацетам библиотеки локального контекста. Дополнительная информация о фацетах приведена на с. 670 и 676.

² Термин «конец файла» в данном контексте означает «конец входных данных», что соответствует интерпретации константы `EOF` в языке С.

Соответственно по умолчанию используется класс трактовок `char_traits<charT>`, где `charT` — тип символов. Стандартная библиотека C++ содержит специализации шаблона `char_traits` для типов символов `char` и `wchar_t`. За дополнительной информацией о трактовках символов обращайтесь к с. 659.

Существуют две специализации класса `basic_ios<>` для двух самых распространенных типов символов:

```
namespace std {
    typedef basic_ios<char>    ios;
    typedef basic_ios<wchar_t> wios;
}
```

Тип `ios` соответствует базовому классу «старой» библиотеки `IOStream`, разработанной AT&T, и может использоваться для обеспечения совместимости со старыми программами C++.

Класс потокового буфера, используемый `basic_ios`, определяется аналогично:

```
namespace std {
    template <class charT,
              class traits = char_traits<charT> >
    class basic_streambuf;
    typedef basic_streambuf<char>    streambuf;
    typedef basic_streambuf<wchar_t> wstreambuf;
}
```

Конечно, шаблоны `basic_istream<>`, `basic_ostream<>` и `basic_iostream<>` тоже параметризуются по типу символов и классу трактовок:

```
namespace std {
    template <class charT,
              class traits = char_traits<charT> >
    class basic_istream;

    template <class charT,
              class traits = char_traits<charT> >
    class basic_ostream;

    template <class charT,
              class traits = char_traits<charT> >
    class basic_iostream;
}
```

По аналогии с другими классами также существуют специализации для двух важнейших типов символов:

```
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_istream<char>    iostream;
typedef basic_istream<wchar_t> wiostream;
}

```

В западном полушарии обычно используются типы *istream* и *ostream*, в целом совместимые со «старыми» потоковыми классами AT&T.

Классы *istream_withassign*, *ostream_withassign* и *iostream_withassign*, включенные в некоторые старые потоковые библиотеки (и производные от *istream*, *ostream* и *iostream* соответственно), не поддерживаются стандартом. Их функциональность реализуется другими средствами (см. с. 615).

Также в библиотеку *IOStream* входят дополнительные классы для выполнения форматированного ввода-вывода при работе с файлами и строками. Эти классы рассматриваются на с. 602 и 619.

Глобальные потоковые объекты

Для потоковых классов определен ряд глобальных потоковых объектов, предназначенных для работы со стандартными каналами ввода-вывода (табл. 13.2). Эти объекты упоминались выше при описании потоков данных с типом символов *char*, однако аналогичные объекты также определены для потоков данных с типом символов *wchar_t*.

Таблица 13.2. Глобальные потоковые объекты

Тип	Имя	Назначение
<i>istream</i>	<i>cin</i>	Читает данные из стандартного канала ввода
<i>ostream</i>	<i>cout</i>	Записывает «нормальные» данные в стандартный канал вывода
<i>ostream</i>	<i>cerr</i>	Записывает сообщения об ошибках в стандартный канал вывода ошибок
<i>ostream</i>	<i>clog</i>	Записывает журнальные данные в стандартный канал вывода журнала
<i>wistream</i>	<i>wcin</i>	Читает данные с расширенной кодировкой символов из стандартного канала ввода
<i>wostream</i>	<i>wcout</i>	Записывает «нормальные» данные с расширенной кодировкой символов в стандартный канал вывода
<i>wostream</i>	<i>wcerr</i>	Записывает сообщения об ошибках с расширенной кодировкой символов в стандартный канал вывода ошибок
<i>wostream</i>	<i>wclog</i>	Записывает журнальные данные с расширенной кодировкой символов в стандартный канал вывода журнала

По умолчанию эти стандартные потоки данных синхронизируются со стандартными потоками данных С. Иначе говоря, стандартная библиотека С++ гарантирует сохранение порядка вывода при смешанном использовании потоков данных С и С++. Перед выводом все буферы стандартных потоков данных С++ очищают буферы соответствующих потоков данных С, и наоборот. Разумеется, синхронизация требует дополнительного времени. Если она не нужна, отключите ее вызовом *sync_with_stdio(false)* перед первой операцией ввода-вывода (см. с. 654).

Заголовочные файлы

Определения потоковых классов распределены по нескольким заголовочным файлам.

- `<iostream>`. Содержит опережающие объявления потоковых классов. Этот заголовочный файл необходим из-за того, что простые опережающие объявления вида `class ostream` теперь не разрешены.
- `<streambuf>`. Содержит определения базового потокового класса с буферизацией (`basic_streambuf<>`).
- `<iostream>`. Содержит определения классов, поддерживающих только ввод (`basic_istream<>`), а также классов с поддержкой ввода и вывода (`basic_iostream<>`)¹.
- `<ostream>`. Содержит определения потокового класса вывода (`basic_ostream<>`).
- `<iostream>`. Содержит объявления глобальных потоковых объектов (таких, как `cin` и `cout`).

Многие из этих заголовочных файлов предназначены для внутренней организации стандартной библиотеки C++. Прикладному программисту обычно достаточно включить файл `<iostream>` в объявление потоковых классов и `<iostream>` или `<ostream>` при непосредственном использовании функций ввода или вывода. Заголовок `<iostream>` следует включать только при использовании стандартных потоковых объектов. В некоторых реализациях в начале работы каждой единицы трансляции, включающей этот заголовок, выполняется фрагмент кода инициализации. Само по себе выполнение этого кода обходится недорого, но при этом приходится загружать соответствующие страницы исполняемого файла, а эта операция может быть довольно дорогостоящей. Как правило, в программу следует включать только заголовки, содержащие абсолютно необходимые объявления. В частности, в заголовочные файлы должен включаться только заголовок `<iostream>`, а соответствующие файлы реализации включают заголовок с полным определением.

Специальные средства работы с потоками данных (параметризованные манипуляторы, файловые и строковые потоки данных) определяются в дополнительных заголовочных файлах (`<iomanip>`, `<fstream>`, `<sstream>` и `<strstream>`). Дополнительная информация об этих заголовках приводится в разделах, посвященных этим специальным средствам.

стандартные операторы << и >>

В С и C++ операторы `<<` и `>>` используются для сдвига битов целых чисел вправо и влево. Классы `basic_istream` и `basic_ostream` перегружают операторы `>>` и `<<` для выполнения стандартных операций ввода-вывода.

¹ На первый взгляд кажется, что объявлять классы с поддержкой ввода и вывода в заголовке `<iostream>` нелогично. Но так как в начале работы каждой единицы трансляции, включающей `<iostream>`, тратится время на инициализацию, объявления для ввода и вывода были выделены в файл `<iostream>`.

Оператор вывода <<

Класс `basic_ostream` (а следовательно, и классы `ostream` и `wostream`) интерпретирует `<<` как оператор вывода. Он перегружает этот оператор для всех базовых типов, включая `char*`, `void` и `bool`.

В соответствии с определением операторов потокового вывода второй аргумент передается в соответствующий поток данных. Иначе говоря, данные пересылаются в направлении «стрелки»:

```
int i = 7;  
std::cout << i;           // Вывод: 7  
  
float f = 4.5;  
std::cout << f;           // Вывод: 4.5
```

Оператор `<<` перегружается так, что второй аргумент может относиться к произвольному типу данных. Это позволяет интегрировать ваши типы данных с системой ввода-вывода. Компилятор следит за тем, чтобы для вывода второго аргумента была вызвана правильная функция. Конечно, эта функция должна преобразовать второй аргумент в последовательность символов, выводимых в поток данных.

В стандартной библиотеке C++ этот механизм также применяется при выводе строк (см. с. 506), битовых полей (см. с. 452) и комплексных чисел (см. с. 518):

```
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)
```

Принципы определения операторов вывода для пользовательских типов данных рассматриваются на с. 625.

Возможность расширения механизма вывода для поддержки пользовательских типов данных является существенным усовершенствованием по сравнению с механизмом ввода-вывода языка С, основанным на использовании функции `printf()`, поскольку не требуется указание типа выводимого объекта. Перегрузка оператора для различных типов гарантирует, что правильная функция вывода будет выбрана автоматически. Оператор `<<` не ограничивается стандартными типами. Таким образом, в распоряжении программиста появляется единый механизм вывода, работающий со всеми типами.

Оператор `<<` также может использоваться для вывода нескольких объектов в одной команде. По общепринятым правилам операторы вывода возвращают свой первый аргумент, то есть результатом выполнения оператора вывода явля-

стся выходной поток данных. Это позволяет объединять вызовы операторов вывода в цепочки вида:

```
std::cout << x << "times" << y << " is " << x * y << std::endl;
```

Цепочечные вызовы оператора << обрабатываются слева направо. Так, в приведенном примере первой будет выполнена следующая команда:

```
std::cout << x
```

Учтите, что это правило не распространяется на порядок вычисления аргументов, — оно определяет только порядок выполнения операторов вывода. Выражение возвращает свой первый operand std::cout. Следующей будет выполнена команда:

```
std::cout << " times "
```

Далее последовательно выводятся объект y, строковый литерал " is " и результат операции x*y. Оператор умножения обладает более высоким приоритетом, чем оператор <<, поэтому выражение x*y не нужно заключать в скобки, однако существуют операторы с более низким приоритетом (например, все логические операторы). Если в данном примере x и y — вещественные числа со значениями 2.4 и 5.1, будет выведена следующая строка:

```
2.4 times 5.1 is 12.24
```

>оператор ввода >>

Класс basic_istream (а следовательно, и классы istream и wistream) интерпретирует >> как оператор ввода. По аналогии с basic_ostream он перегружает этот оператор для всех базовых типов, включая char*, void и bool. Потоковые операторы ввода сохраняют прочитанное значение во втором аргументе. Как и в случае с оператором <<, данные пересыпаются в направлении «стрелки»:

```
int i;  
std::cin >> i; // Читает int из стандартного ввода и сохраняет его в i  
  
float f;  
std::cin >> f; // Читает float из стандартного ввода и сохраняет его в f
```

Обратите внимание на модификацию второго аргумента. Чтобы это было возможно, второй аргумент передается по неконстантной ссылке.

Оператор ввода, как и оператор вывода, может перегружаться для произвольных типов данных и может вызываться «по цепочке»:

```
float f;  
std::complex<double> c;  
  
std::cin >> f >> c;
```

По умолчанию начальные пропуски игнорируются, хотя этот режим можно отключить (см. с. 600).

Ввод-вывод специальных типов

Кроме базовых типов стандартные операторы ввода-вывода также определены для типов `bool`, `char*` и `void*`. Кроме того, они могут расширяться для базовых типов.

Тип `bool`

По умолчанию логические величины вводятся и выводятся в численном представлении: `false` соответствует числу 0, а `true` соответствует 1. При чтении логических данных значения, отличные от 0 и 1, считаются ошибочными. В этом случае устанавливается бит `ios::failbit`, что может привести к выдаче соответствующего исключения (см. с. 576).

Для потока данных также можно включить режим форматирования, в котором логические величины вводятся и выводятся в виде символьных строк (см. с. 592). При этом приходится учитывать проблему интернационализации: без специального локального контекста используются строки "true" и "false". В других локальных контекстах могут применяться другие строки. Например, объект локального контекста немецкого языка ассоциирует логические значения со строками "wahr" и "falsch". За подробностями обращайтесь к с. 671.

Типы `char` и `wchar_t`

При чтении типов `char` или `wchar_t` оператором `>>` начальные пропуски по умолчанию игнорируются. Чтобы оператор читал любые символы (как пропуски, так и обычные символы), либо сбросьте флаг `skipws` (см. с. 600), либо воспользуйтесь функцией `get()` (см. с. 582).

Тип `char*`

С-строки (типа `char*`) вводятся по словам. Иначе говоря, при чтении С-строки начальные пропуски по умолчанию игнорируются, а чтение продолжается до следующего пропуска или конца файла. Игнорированием начальных пропусков можно управлять при помощи флага `skipws` (см. с. 600).

Из этого описания следует, что строка может иметь произвольную длину. Некоторые программисты С ошибочно считают, что максимальная длина строки ограничивается 80 символами. На самом деле такого ограничения не существует. Следовательно, программист должен сам следить за своевременным завершением слишком длинных строк. Для этого следует *всегда* задавать максимальную длину читаемой строки. Обычно это делается примерно так:

```
char buffer[81];      // 80 символов и \0
std::cin >> std::setw(81) >> buffer;
```

Манипулятор `setw()` и соответствующий параметр потока данных подробно рассматриваются на с. 593.

Тип `string` стандартной библиотеки C++ (см. главу 11) автоматически растет по мере необходимости. Гораздо проще и безопаснее использовать класс `string` вместо `char*`. Кроме того, он поддерживает удобную функцию построчного ввода (см. с. 475). Постарайтесь обойтись без С-строк и используйте обычные строковые классы.

Тип `void*`

Операторы `>>` и `<<` также позволяют вывести указатель и снова прочитать его. Если оператору вывода передается параметр типа `void*`, то оператор выводит адрес в формате, зависящем от реализации. Например, следующая команда выводит содержимое С-строки и ее адрес:

```
char* cstring = "hello";  
  
std::cout << "string \" " << cstring << "\" is located at address: "  
      << static_cast<void*>(cstring) << std::endl;
```

Результат выполнения этой команды выглядит так:

```
string "hello" is located at address: 0x10000018
```

Этот адрес можно прочитать оператором ввода. Тем не менее следует учитывать, что в общем случае адрес является временной величиной. При новом запуске программы объекту может быть присвоен совершенно иной адрес. В частности, ввод-вывод адресов может применяться в программах, обменивающихся адресами для идентификации объектов или использующих общую память.

Потоковые буферы

Операторы `<<` и `>>` могут использоваться соответственно для прямого чтения из потокового буфера и записи в потоковый буфер. Вероятно, это самый быстрый способ копирования файлов с применением потокового ввода-вывода C++. Пример приведен на с. 575.

Пользовательские типы

В принципе механизм ввода-вывода легко расширяется для пользовательских типов. Тем не менее, с учетом всех доступных вариантов форматирования данных и возможных ошибок, это не так просто, как может показаться. Применение стандартного механизма ввода-вывода для пользовательских типов подробно рассматривается на с. 625.

СОСТОЯНИЕ ПОТОКА ДАННЫХ

Потоки данных обладают определенным состоянием. В частности, состояние показывает, успешно или нет была выполнена операция ввода-вывода, а если нет — то почему.

Константы состояния потока данных

Общее состояние потока данных определяется несколькими флагами, представленными константами типа `iostate` (табл. 13.3). Тип `iostate` определяется в классе `ios_base`. Конкретный тип констант зависит от реализации (иначе говоря, стандарт не указывает, является тип `iostate` перечислением, определением целочисленного типа или специализацией класса `bitset`).

Таблица 13.3. Константы типа `iostate`

Константа	Описание
<code>goodbit</code>	Нормальное состояние потока данных, другие биты не установлены
<code>eofbit</code>	Обнаружен признак конца файла
<code>failbit</code>	Ошибка; операция ввода-вывода завершилась неудачно
<code>badbit</code>	Фатальная ошибка, неопределенное состояние потока данных

Бит `goodbit` по определению равен 0. Таким образом, установка флага `goodbit` означает, что все остальные биты также равны 0. Имя `goodbit` выбрано не совсем удачно, поскольку нормальное состояние потока данных обозначается не установкой отдельного бита, а сбросом всех остальных битов.

Основное отличие флагов `failbit` и `badbit` состоит в том, что `badbit` обозначает более серьезную ошибку.

- флаг `failbit` устанавливается в том случае, если операция завершилась неудачно, но состояние потока данных позволяет продолжить работу. Обычно этот флаг устанавливается при ошибках форматирования в процессе чтения — например, если программа пытается прочитать целое число, а следующий символ является буквой.
- флаг `badbit` указывает на неработоспособность потока данных или потерю данных, например, при установке указателя в файловом потоке данных перед началом файла.

Флаг `eofbit` обычно устанавливается вместе с `failbit`, поскольку признак конца файла проверяется и обнаруживается при попытке чтения за концом файла. После чтения последнего символа флаг `eofbit` не устанавливается, он устанавливается *вместе* с флагом `failbit` при следующей попытке чтения символа.

В некоторых старых реализациях также определен флаг `hardfail`. Стандартом этот флаг не поддерживается.

Константы флагов определяются не глобально, а в классе `ios_base`. Это означает, что они всегда должны использоваться с указанием области видимости или с конкретным объектом. Пример:

```
std::ios_base::eofbit
```

Конечно, с таким же успехом можно использовать класс, производный от `ios_base`. В старых реализациях эти константы определялись в классе `ios`. Поскольку тип `ios` является производным от `ios_base`, а его имя вводится быстрее, в программах часто встречаются конструкции вида:

```
std::ios::eofbit
```

Флаги определены в базовом классе `basic_ios`, поэтому они присутствуют во всех объектах типов `basic_istream` и `basic_ostream`. С другой стороны, потоковые буферы состояния не имеют. Один буфер может совместно использоваться несколькими потоковыми объектами, поэтому флаги представляют только состояние потока данных в результате последней операции и только в случае,

если перед операцией поток данных был приведен в состояние `goodbit`. В противном случае может оказаться, что флаги были установлены одной из предшествующих операций.

Функции для работы с состоянием потока данных

В табл. 13.4 перечислены функции для работы с текущим состоянием потока данных.

Таблица 13.4. Функции для работы с состоянием потока данных

Функция	Описание
<code>good()</code>	Возвращает <code>true</code> , если поток данных находится в нормальном состоянии (то есть при установке флага <code>goodbit</code>)
<code>eof()</code>	Возвращает <code>true</code> при обнаружении признака конца файла (установка флага <code>eofbit</code>)
<code>fail()</code>	Возвращает <code>true</code> при обнаружении ошибки (установка флага <code>failbit</code> или <code>badbit</code>)
<code>bad()</code>	Возвращает <code>true</code> при обнаружении фатальной ошибки (установка флага <code>badbit</code>)
<code>rdstate()</code>	Возвращает установленные флаги
<code>clear()</code>	Сбрасывает все флаги
<code>clear(state)</code>	Сбрасывает все флаги и устанавливает флаги, содержащиеся в <code>state</code>
<code>setstate(state)</code>	Устанавливает флаги, содержащиеся в <code>state</code>

Первые четыре функции проверяют состояние отдельных флагов и возвращают логическую величину. Учтите, что функция `fail()` возвращает `true` при установке как `failbit`, так и `badbit`. Хотя такое поведение объясняется в основном историческими причинами, у него есть определенные достоинства — для выявления ошибки достаточно проверить всего одно условие.

Также существуют более общие функции для проверки и изменения состояния флагов. При вызове функции `clear()` без параметров сбрасываются все флаги ошибок (в том числе и `eofbit`):

```
// Сброс всех флагов ошибок (включая eofbit)
strm.clear();
```

Если функция `clear()` вызывается с передачей параметра, то состояние потока данных изменяется в соответствии с переданным значением; переданные флаги устанавливаются для потока данных, а остальные флаги сбрасываются. У этого правила есть единственное исключение: при отсутствии потокового буфера (когда `rdbuf()==0`) флаг `badbit` устанавливается всегда; за подробностями обращайтесь к с. 613.

Следующий пример показывает, как установить и сбросить флаг `failbit`:

```
// Проверка установки флага failbit
if (strm.rdstate() & std::ios::failbit) {
    std::cout << "failbit was set" << std::endl;
```

```
// Сброс только флага failbit
strm.clear(strm.rdstate() & ~std::ios::failbit);
}
```

В этом примере используются поразрядные операторы `&` и `~`. Оператор `~` возвращает поразрядное дополнение своего аргумента. Следовательно, показанное ниже выражение возвращает временное значение, в котором установлены все биты, кроме `failbit`:

```
~ios::failbit
```

Оператор `&` возвращает результат поразрядного объединения своих операндов. В результате операции устанавливаются только биты, установленные в обоих операндах. При поразрядном объединении всех текущих установленных флагов (`rdstate()`) со всеми установленными битами, кроме `failbit`, бит `failbit` сбрасывается, а значения всех остальных битов сохраняются.

Потоки данных можно настроить так, чтобы при установке флагов функциями `clear()` и `setstate()` генерировались исключения (см. с. 576).

Также следует помнить о необходимости явного сброса битов ошибок. Язык C позволял продолжить чтение символов после ошибки форматирования. Например, если функции `scanf()` не удавалось прочитать целое число, программа могла ввести оставшиеся символы. Таким образом, операция чтения завершалась неудачей, но входной поток данных оставался в нормальном состоянии. В C++ дело обстоит иначе. При установке бита `failbit` все последующие операции с потоком данных игнорируются до тех пор, пока флаг `failbit` не будет сброшен программой.

Вообще говоря, установленные флаги всего лишь отражают события, которые происходили в прошлом. Наличие установленного флага после какой-либо операции не всегда означает, что именно эта операция привела к его установке. Флаг также мог быть установлен еще до выполнения этой операции. Следовательно, если вы собираетесь проверять результат выполнения операции по состоянию флагов, перед ее выполнением следует привести поток данных к состоянию `goodbit` (если нет полной уверенности в том, что он уже находится в этом состоянии). Кроме того, после сброса флагов операция может вернуть другой результат. Например, даже если флаг `eofbit` в результате операции был установлен, это еще не означает, что после сброса `eofbit` (и всех остальных установленных битов) флаг будет снова установлен при повторном выполнении операции, поскольку между вызовами файл может увеличиться.

Состояние потока данных и логические условия

В табл. 13.5 приведены две операторные функции, предназначенные для проверки состояния потоков данных в логических выражениях.

Таблица 13.5. Потоковые операторы для логических выражений

Функция	Описание
<code>operator void*()</code>	Проверяет нормальное состояние потока данных (эквивалент <code>!fail()</code>)
<code>operator ! ()</code>	Проверяет ошибочное состояние потока данных (эквивалент <code>fail()</code>)

Оператор `void*()` обеспечивает короткую и наглядную запись проверки состояния потока данных в управляющих структурах:

```
// Пока стандартный поток ввода находится в нормальном состоянии...
while (std::cin) {
    ...
}
```

Проверка логических условий в управляющих структурах не требует прямого преобразования к `bool`. Достаточно уникального преобразования к целому типу (например, `int` или `char`) или к типу указателя. Для чтения объекта и проверки результата в одной команде часто требуется преобразование к `void*`:

```
if (std::cin >> x) {
    // Чтение x выполнено успешно
    ...
}
```

Как было показано ранее, следующее выражение возвращает `cin`:

```
std::cin >> x
```

Следовательно, после чтения `x` команда эквивалентна такой команде:

```
if (std::cin) {
    ...
}
```

Объект `cin` часто используется при проверке условий; оператор `void*` этого объекта возвращает признак наличия ошибки потока данных.

Этот прием обычно применяется в циклах, выполняющих чтение и обработку объектов:

```
// Пока удается прочитать obj
while (std::cin >> obj) {
    // Обработка obj (в данном случае - простой вывод)
    std::cout << obj << std::endl;
}
```

В этом фрагменте легко угадывается классический синтаксис фильтров C, примененный к объектам C++. Цикл завершается при установке флага `failbit` или `badbit`. Эти флаги устанавливаются при возникновении ошибки или достижении конца файла (попытка чтения за концом файла приводит к установке флага `eofbit` или `badbit`, как показано на с. 572). По умолчанию оператор `>>` игнорирует начальные пропуски. Обычно это вполне устраивает программиста, но если `obj` относится к типу `char`, то пропуски могут оказаться существенными. В этом случае в реализации фильтра можно использовать функции `get()` и `put()` потоковых классов (см. с. 582), а еще лучше — итератор `istreambuf_iterator` (см. с. 640).

Оператор `!` проверяет обратное условие. В соответствии со своим определением он возвращает `true`, если для потока данных установлен бит `failbit` или `badbit`. Возможный вариант использования:

```
if (! std::cin) {
    // Поток cin находится в ошибочном состоянии
    ...
}
```

Как и в случае с неявным преобразованием к логическому типу, этот оператор часто используется для совмещения чтения с проверкой результата:

```
if (!(std::cin >> x)) {
    // Попытка чтения завершилась неудачей
    ...
}
```

Следующее выражение возвращает объект `cin`, к которому применяется оператор `!`:

```
std::cin >> x
```

Выражение после оператора `!` необходимо заключить в круглые скобки из-за относительного приоритета операторов; без круглых скобок оператор `!` будет выполнен первым. Другими словами, следующие два выражения эквивалентны:

```
!std::cin >> x
(!std::cin) >> x
```

Вряд ли это именно то, к чему стремился программист.

Хотя описываемые операторы очень удобны, следует обратить внимание на одну странность: двойное отрицание *не означает* возвращения к исходному объекту:

- `cin` — потоковый объект класса `istream`;
- `!!cin` — логическая величина, описывающая состояние объекта `cin`.

Существует мнение, что преобразование к логическому значению не соответствует принципам хорошего стиля программирования. Функции типа `fail()` обычно делают программу более понятной:

```
std::cin >> x;
if (std::cin.fail()) {
    ...
}
```

Состояние потока данных и исключения

Механизм обработки исключений был включен в C++ для выявления ошибок и особых ситуаций (см. с. 31). Тем не менее это было сделано уже после того, как потоки данных получили широкое распространение. Чтобы сохранить совместимость, потоки данных по умолчанию не генерируют исключений. Однако для каждого флага состояния стандартизованных потоков данных можно указать, должна ли установка этого флага сопровождаться выдачей исключения. Для этой цели используется функция `exceptions()` (табл. 13.6).

Таблица 13.6. Функции управления исключениями

Функция	Описание
exceptions (флаги)	Определяет флаги, при установке которых должно генерироваться исключение
exceptions()	Возвращает флаги, при установке которых должно генерироваться исключение

При вызове без аргументов функция `exceptions()` возвращает текущие флаги, при установке которых генерируются исключения. Если функция возвращает `goodbit`, исключения не генерируются. Этот режим используется по умолчанию для поддержания совместимости. Если функция `exceptions()` вызывается с аргументом, то сразу же после установки соответствующего флага состояния будет выдано исключение.

Следующий пример настраивает поток данных так, чтобы при установке любого флага генерировалось исключение:

```
// Генерировать исключение при любой "ошибке"
strm.exceptions (std::ios::eofbit | std::ios::failbit |
                 std::ios::badbit);
```

Если аргумент равен 0 или `goodbit`, исключения не генерируются:

```
// Не генерировать исключения
strm.exceptions (std::ios::goodbit);
```

Исключения генерируются при установке соответствующих флагов после вызова `clear()` или `setstate()`. Это происходит даже в том случае, если флаг был установлен ранее:

```
// Вызов генерирует исключение, если флаг failbit был установлен при входе
strm.exceptions (std::ios::failbit);

...
// Следующая команда генерирует исключение (даже если флаг failbit
// был установлен ранее)
strm.setstate (std::ios::failbit);
```

Генерируемые исключения являются объектами класса `std::ios_base::failure`, производного от класса `exception` (см. с. 43).

```
namespace std {
    class ios_base::failure : public exception {
        public:
            // Конструктор
            explicit failure (const string& msg);

            // Деструктор
            virtual ~failure();

            // Возврат информации об исключении
    };
}
```

```
    virtual const char* what() const;
}:
```

К сожалению, стандарт не требует, чтобы объект исключения содержал какую-либо информацию об ошибочном потоке данных или о типе ошибки. Существует единственный переносимый способ получения информации об ошибке — сообщение, возвращаемое функцией `what()`. Впрочем, переносим только *вызов* `what()`, но не возвращаемая строка. Если требуется дополнительная информация, программист должен сам заботиться о ее получении.

Из этого поведения следует, что обработка исключений в большей степени ориентируется на непредвиденные ситуации. Собственно, поэтому она называется обработкой *исключений*, а не обработкой *ошибок*. Ожидаемые ошибки (например, ошибки форматирования при вводе данных пользователем) считаются «нормальными», а для их обработки лучше использовать флаги состояния.

Основная область применения потоковых исключений — чтение предварительно отформатированных данных (например, автоматически генерированных файлов). Но даже в этом случае при обработке исключений возникают проблемы. Например, если данные читаются до конца файла, вы не сможете генерировать исключения ошибок без того, чтобы не получить исключение конца файла. Дело в том, что при обнаружении конца файла также устанавливается бит `failbit` (признак неудачи при чтении объекта). Чтобы отличить конец файла от ошибки ввода, придется дополнительно проверить состояние потока данных.

Следующий пример показывает, как это делается. Функция `readAndProcessSum` читает вещественные числа из потока данных до тех пор, пока не достигнут конец файла. Затем возвращается сумма прочитанных вещественных чисел:

```
// io/sum1a.cpp
#include <iostream>

namespace MyLib {
    double readAndProcessSum (std::istream& strm)
    {
        using std::ios;
        double value, sum;

        // Сохранение текущего состояния флагов исключений
        ios::iostate oldExceptions = strm.exceptions();

        /* Выдача исключений при установке флагов failbit и badbit
         * - ВНИМАНИЕ: флаг failbit также устанавливается
         *   при достижении конца файла
         */
        strm.exceptions (ios::failbit | ios::badbit);

        try {
            /* Пока поток находится в нормальном состоянии
             * - прочитать значение и прибавить его к сумме
             */
        }
```

```
sum = 0;
while (strm >> value) {
    sum += value;
}
catch (...) {
    /* Если исключение произошло не из-за достижения конца файла.
     * - восстановить старую маску исключений
     * - перезапустить исключение
     */
    if (!strm.eof()) {
        strm.exceptions(oldExceptions); // Восстановление маски
        throw;                         // Перезапуск исключения
    }
}

// Восстановление старой маски исключений
strm.exceptions (oldExceptions);

// Возврат суммы
return sum;
}
```

Сначала функция сохраняет состояние маски исключений в переменной `oldExceptions`, чтобы восстановить ее позднее. Затем поток данных настраивается на выдачу исключений при требуемых условиях. Функция читает значения в цикле и суммирует их до тех пор, пока поток данных остается в нормальном состоянии. При достижении конца файла состояние потока данных перестает быть нормальным, и он генерирует исключение (хотя для флага `eofbit` исключение не генерировалось). Дело в том, что конец файла обнаруживается при неудачной попытке чтения новых данных, которая также устанавливает флаг `failbit`. Чтобы избежать выдачи исключения при достижении конца файла, мы организуем локальный перехват исключений и проверяем состояние потока данных функцией `eof()`. Исключение передается дальше только в том случае, если `eof()` возвращает `false`.

Следует помнить, что восстановление исходной маски исключений также может привести к выдаче исключений. Функция `exceptions()` генерирует исключение в том случае, если соответствующий флаг уже установлен для потока данных. Поэтому если поток данных генерировал исключения для флага `eofbit`, `failbit` или `badbit` при входе в функцию, эти исключения также будут переданы вызывающей стороне.

Ниже приведен простой пример вызова этой функции из `main()`.

```
// io/summain.cpp
#include <iostream>
#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;
    }

    // Вывод суммы
    cout << "sum: " << sum << endl;
}

```

Но возникает вопрос: насколько оправданы эти хлопоты? Проще работать с потоками данных, которые не выдают исключений. В этом случае программа сама генерирует исключение при обнаружении ошибки. Дополнительным достоинством такого решения является возможность работы с пользовательскими сообщениями об ошибках и классами ошибок:

```

// io/sum2a.cpp
#include <iostream>

namespace MyLib {
    double readAndProcessSum (std::istream& strm)
    {
        double value, sum;

        /* Пока поток остается в нормальном состоянии
         * - прочитать очередное значение и прибавить его к сумме
         */
        sum = 0;
        while (strm >> value) {
            sum += value;
        }

        if (!strm.eof()) {
            throw std::ios::failure
                ("input error in readAndProcessSum()");
        }
    }
}

```

```
    }

    // Возврат суммы
    return sum;
}

}
```

Выглядит гораздо проще, не правда ли? Этой версии функции нужен заголовок `<string>`, поскольку конструктор класса `failure` получает в аргументе ссылку на константный объект `string`. Для конструирования объекта этого типа потребуется определение, а для передачи объявления достаточно заголовка `<iostream>`.

Стандартные функции ввода-вывода

Вместо стандартных операторов `>>` и `<<` для чтения из потока данных и записи в поток данных можно использовать ряд других функций, представленных в этом разделе.

Эти функции предназначены для чтения и записи «неформатированных» данных (в отличие от операторов `<<` и `>>`, которые читают и записывают «форматированные» данные). Функции при чтении никогда не игнорируют начальные пропуски (в отличие от операторов, которые по умолчанию начальные пропуски игнорируют). Кроме того, в них используется другой принцип обработки исключений: флаг `badbit` устанавливается, если функция генерирует исключение, причем не важно, кто является источником исключения — сама вызванная функция или исключение генерируется в результате установки флага состояния (см. с. 576). Если в маске исключений установлен флаг `badbit`, исключение передается дальше. Тем не менее функции неформатированного вывода, как и функции форматированного вывода, создают объект `sentry` (см. с. 631).

Для передачи количества символов в этих функциях используется тип `streamsize`:

```
namespace std {
    typedef ... streamsize;
    ...
}
```

Тип `streamsize` обычно представляет собой знаковую версию `size_t`. Тип является знаковым, потому что он также используется для передачи отрицательных значений.

Функции ввода

В следующих определениях `istream` обозначает потоковый класс, используемый для чтения. Это может быть класс `istream`, `wistream` или любая другая специализация класса шаблона `basic_istream`. Параметр `char` обозначает соответствующий тип символов (`char` для `istream`, `wchar_t` для `wistream`). Другие типы и значения, выводимые курсивом, зависят от определения типа символов или класса трактовок, связанного с потоком данных.

В стандартную библиотеку C++ входят несколько функций потоковых классов для чтения последовательностей символов. Возможности этих функций сравниваются в табл. 13.7.

Таблица 13.7. Функции чтения последовательностей символов

Функция	Признак конца чтения	Количество символов	Присоединение завершителя	Возвращаемый тип
get(s, num)	Новая строка (без включения) или конец файла	До num-1	Да	istream
get(s, num, t)	t (без включения) или конец файла	До num-1	Да	istream
getline(s, num)	Новая строка (с включением) или конец файла	До num-1	Да	istream
getline(s, num, t)	t (с включением) или конец файла	До num-1	Да	istream
read(s, num)	Конец файла	num	Нет	istream
readsome(s, num)	Конец файла	До num	Нет	streamsize

int istream::get ()

- Читает следующий символ.
- Возвращает прочитанный символ или *EOF*.
- В общем случае возвращаемое значение относится к типу *traits::int_type*, а *EOF* – величина, возвращаемая при вызове *traits::eof()*. Для *istream* это соответственно тип *int* и константа *EOF*. Следовательно, для *istream* эта функция соответствует функциям *getchar()* и *getc()* языка C.
- Возвращаемое значение не обязательно относится к типу символов потока данных; оно также может относиться к типу с более широким диапазоном значений. Без этого было бы невозможно отличить *EOF* от символа с соответствующим значением.

istream& istream::get (char& c)

- Присваивает следующий символ аргументу *c*.
- Возвращает объект потока данных, по состоянию которого можно проверить, успешно ли выполнено чтение.

istream& istream::get (char str, streamsize count)*

istream& istream::get (char str, streamsize count, char delim)*

- Обе формы читают до *count-1* символов в строку *str*.
- Первая форма завершает чтение, если следующий читаемый символ является символом новой строки соответствующей кодировки. Для *istream* это символ *\n*, а для *wistream* – символ *wchar_t('\n')* (см. с. 663). В общем случае используется символ *widen('\n')* (см. с. 602).

- Вторая форма завершает чтение, если следующий читаемый символ является разделителем *delim*.
- Обе формы возвращают объект потока данных, по состоянию которого можно проверить, успешно ли выполнено чтение.
- Завершающий символ (*delim*) не читается.
- Символ завершения строки прерывает чтение.
- Перед вызовом необходимо убедиться в том, что размер *str* достаточен для хранения *count* символов.

istream& istream::getline (char str, streamsize count)*

istream& istream::getline (char str, streamsize count, char delim)*

Обе формы идентичны предыдущим функциям *get()* со следующими исключениями:

- чтение завершается не *перед* символом новой строки или *delim* соответственно, а *включая* этот символ, то есть символ новой строки или *delim* будет прочитан, если он встречается среди *count-1* символов, но он *не сохраняется* в *str*;
- если прочитанная строка содержит более *count-1* символов, функции устанавливают флаг *failbit*.

istream& istream::read (char str, streamsize count)*

- Читает *count* символов в строку *str*.
- Возвращает объект потока данных, по состоянию которого можно проверить, успешно ли выполнено чтение.
- Стока *str* *не завершается* автоматически символом завершения строки.
- Перед вызовом необходимо убедиться в том, что размер *str* достаточен для хранения *count* символов.
- Обнаружение признака конца файла в процессе чтения считается ошибкой, для которой устанавливается бит *failbit* (в добавок к флагу *eofbit*).

streamsize istream::readsome (char str, streamsize count)*

- Читает до *count* символов в строку *str*.
- Возвращает количество прочитанных символов.
- Стока *str* *не завершается* автоматически символом завершения строки.
- Перед вызовом необходимо убедиться в том, что размер *str* достаточен для хранения *count* символов.
- В отличие от функции *read()* функция *readsome()* читает из потокового буфера все доступные символы (при помощи функции *in_avail()* класса буфера). Например, она может использоваться в ситуациях, когда ввод поступает с клавиатуры или от других процессов, поэтому ожидание нежелательно. Обнаружение конца файла не считается ошибкой, а биты *eofbit* и *failbit* не устанавливаются.

`streamsize istream::gcount () const`

- Возвращает количество символов, прочитанных последней операцией *неформатированного* ввода.

`istream& istream::ignore ()`

`istream& istream::ignore (streamsize count)`

`istream& istream::ignore (streamsize count, int delim)`

- Все формы извлекают символы из потока данных и игнорируют их.
- Первая форма игнорирует один символ.
- Вторая форма игнорирует до *count* символов.
- Третья форма игнорирует до *count* символов и прекращает работу тогда, когда будет извлечен и проигнорирован символ *delim*.
- Если значение *count* равно `std::numeric_limits<std::streamsize>::max()`, то есть максимальному значению типа `std::streamsize` (см. с. 72), функция игнорирует все символы до тех пор, пока не будет обнаружен ограничитель *delim* или конец файла.
- Все формы возвращают объект потока данных.

- Примеры:

- игнорирование остатка строки:

```
cin.ignore(numeric_limits<std::streamsize>::max(), '\n');
```

- игнорирование всех оставшихся символов `cin`:

```
cin.ignore(numeric_limits<std::streamsize>::max());
```

`int istream::peek ()`

- Возвращает следующий считываемый из потока данных символ без его извлечения. Символ считывается следующей операцией чтения (если не изменится позиция ввода).
- Если дальнейшее чтение невозможно, возвращает *EOF*.
- *EOF* — значение, возвращаемое `traits::eof()`. Для класса `istream` это константа `EOF`.

`istream& istream::unget ()`

`istream& istream::putback (char c)`

- Обе функции возвращают в поток данных последний считанный символ, чтобы он был считан следующей операцией чтения (если не изменится позиция ввода).
- Различия между функциями `unget` и `pushback()` заключаются в том, что `pushback()` проверяет, был ли передаваемый символ с последним считанным символом.
- Если символ не удается вернуть или функция `pushback()` пытается вернуть другой символ, устанавливается флаг `badbit`, что может привести к выдаче соответствующего исключения (см. с. 576).
- Максимальное количество символов, которые могут быть возвращены в поток данных этими функциями, зависит от реализации. По стандарту гаранти-

рованно работает только один вызов этих функций между двумя операциями чтения. Следовательно, только этот вариант может считаться переносимым.

При чтении С-строк описанные здесь функции безопаснее оператора `>>`, поскольку они требуют явной передачи максимального размера читаемой строки. Хотя количество читаемых символов можно ограничить и при использовании оператора `>>` (см. с. 593), об этом часто забывают.

Вместо использования потоковых функций ввода часто удобнее работать с потоковым буфером напрямую. Функции потоковых буферов позволяют эффективно читать отдельные символы или последовательности символов без затрат на конструирование объектов `sentry` (см. с. 631). Интерфейс потоковых буферов более подробно рассматривается на с. 636. Также можно воспользоваться шаблонным классом `istreambuf_iterator`, предоставляющим итераторный интерфейс к потоковому буферу (см. с. 638).

Функции `tellg()` и `seekg()` предназначены для изменения текущей позиции чтения. В основном они используются при работе с файлами, поэтому их описание откладывается до с. 609.

Функции вывода

В следующих определениях `ostream` обозначает потоковый класс, используемый для записи. Это может быть класс `ostream`, `wostream` или любая другая специализация класса шаблона `basic_ostream`. Параметр `char` обозначает соответствующий тип символов (`char` для `ostream`, `wchar_t` для `wostream`). Другие типы и значения, выводимые курсивом, зависят от определения типа символов или класса трактовок, связанного с потоком данных.

`ostream& ostream::put (char c)`

- Записывает аргумент `c` в поток данных.
- Возвращает объект потока данных, по состоянию которого можно проверить, успешно ли выполнена запись.

`ostream& ostream::write (const char* str, streamsize count)`

- Записывает `count` символов строки `str` в поток данных.
- Возвращает объект потока данных, по состоянию которого можно проверить, успешно ли выполнена запись.
- Символ завершения строки *не останавливает* запись и выводится вместе с остальными символами.
- Перед вызовом необходимо убедиться в том, что `str` содержит не менее `count` символов, иначе вызов приводит к непредсказуемым последствиям.

`ostream& ostream::flush ()`

Очищает потоковые буфера (принудительная запись всех буферизованных данных на устройство или в канал ввода-вывода, с которым связан буфер).

Функции `tellp()` и `seekp()` предназначены для изменения текущей позиции записи. В основном они используются при работе с файлами, поэтому их описание откладывается до с. 609.

По аналогии с функциями ввода иногда бывает удобно работать с потоковым буфером напрямую или воспользоваться шаблонным классом `ostreambuf_iterator` для неформатированного вывода. Функции неформатированного вывода не дают особых преимуществ, если не считать того, что они могут обеспечивать синхронизацию в многопоточных¹ средах с использованием объектов `sentry`. За дополнительной информацией обращайтесь к с. 655.

Пример использования

Классический фильтр, который просто выводит в выходной поток данных все прочитанные символы, выглядит на C++ так:

```
// io/charcat1.cpp
#include <iostream>
using namespace std;

int main()
{
    char c;

    // Пока удаётся прочитать символ
    while (cin.get(c)) {
        // Вывести прочитанный символ в выходной поток
        cout.put(c);
    }
}
```

При каждом вызове `cin.get(c)` следующий символ присваивается переменной `c`, которая передается по ссылке. Функция `get()` возвращает объект потока данных; таким образом, условие `while` остается истинным до тех пор, пока поток данных находится в нормальном состоянии².

Чтобы повысить эффективность программы, можно работать напрямую с потоковыми буферами. На с. 641 приведена версия этого примера, в которой при вводе-выводе используются буферные итераторы, а в версии на с. 656 все входные данные копируются одной командой.

Манипуляторы

Потоковые манипуляторы упоминались на с. 561. Это объекты, которые при применении к ним стандартных операторов ввода-вывода изменяют поток дан-

¹ В данном случае речь идет не об описываемых в этой главе *потоках данных* (*streams*), а о *потоках выполнения* (*threads*). – Примеч. перев.

² Такой интерфейс лучше классического интерфейса фильтров в языке С. В С вам пришлось бы использовать функцию `getchar()` или `getc()`, которая возвращает как следующий символ, так и признак конца файла. В этом случае возвращаемое значение пришлось бы обрабатывать как тип `int`, чтобы отличить символ от признака конца файла.

ных. В табл. 13.8 перечислены основные манипуляторы, определенные в заголовочных файлах `<iostream>` и `<ostream>`.

Таблица 13.8. Манипуляторы, определенные в заголовках `<iostream>` и `<ostream>`

Манипулятор	Класс	Описание
<code>flush</code>	<code>basic_ostream</code>	Принудительный вывод выходного буфера на устройство
<code>endl</code>	<code>basic_ostream</code>	Запись символа новой строки в буфер и принудительный вывод выходного буфера на устройство
<code>ends</code>	<code>basic_ostream</code>	Запись символа завершения строки в буфер
<code>ws</code>	<code>basic_istream</code>	Чтение с игнорированием пропусков

Помимо перечисленных манипуляторов, существуют и другие, например, предназначенные для смены формата ввода-вывода. Они представлены на с. 589.

Принципы работы манипуляторов

Реализация манипуляторов основана на очень простом приеме, который не только упрощает управление потоками данных, но и наглядно демонстрирует мощь механизма перегрузки функций. Манипуляторы представляют собой обычные функции, передаваемые операторам ввода-вывода в аргументах. Оператор вызывает переданную функцию. Например, оператор вывода класса `ostream` перегружается примерно так¹:

```
ostream& ostream::operator << ( ostream& (*op)(ostream&) )
{
    // Вызов функции, передаваемой в параметре, с аргументом-потоком
    return (*op)(*this);
}
```

Аргумент `op` представляет собой указатель на функцию. Точнее говоря, это функция, которая получает и возвращает поток данных `ostream` (предполагается, что возвращается тот же объект `ostream`, который был получен при вызове). Если второй operand оператора `<<` является такой функцией, то при ее вызове в аргументе передается первый operand оператора `<<`.

На первый взгляд описание кажется очень сложным, но в действительности все относительно просто. Следующий пример поможет лучше разобраться в происходящем. Манипулятор (то есть функция) `endl()` для объекта `ostream` реализуется примерно так:

```
std::ostream& std::endl ( std::ostream& strm )
{
    // Запись признака конца строки
    strm.put('\n');
```

¹ Конкретная реализация выглядит несколько сложнее, потому что ей приходится конструировать объект `sentry`, а также потому, что она оформляется в виде шаблона.

```

// Принудительный вывод выходного буфера
strm.flush();

// Возвращение strm для организации целочечных вызовов
return strm;
}

```

Манипулятор используется в выражениях вида

```
std::cout << std::endl;
```

Для потока данных cout вызывается оператор <<, которому во втором операнде передается функция endl(). Реализация оператора << преобразует этот вызов в вызов переданной функции, которой в качестве аргумента передается объект потока данных:

```
std::endl(std::cout)
```

Чтобы добиться эффекта «вывода» манипулятора, можно просто использовать это выражение. Более того, у функциональной записи есть свои преимущества — она не требует указания пространства имен:

```
endl(std::cout)
```

Это возможно благодаря тому, что функции ищутся в том пространстве имен, в котором определены их аргументы (см. с. 33).

Поскольку потоковые классы оформлены в виде шаблонов, параметризованных по типу символов, настоящая реализация endl() выглядит примерно так:

```

template<class charT, class traits>
std::basic_ostream<charT,traits>&
std::endl (std::basic_ostream<charT,traits>& strm)
{
    strm.put(strm.widen('\n'));
    strm.flush();
    return strm;
}

```

Функция widen() преобразует символ новой строки к кодировке, используемой в потоке данных. За подробностями обращайтесь к с. 601.

В стандартную библиотеку C++ также включены параметризованные манипуляторы (то есть манипуляторы, которым при вызове передаются аргументы). Принципы работы этих манипуляторов зависят от реализации; не существует стандартных правил определения пользовательских параметризованных манипуляторов.

Стандартные параметризованные манипуляторы определяются в заголовочном файле `<iomanip>`. Если вы собираетесь использовать их, в программу необходимо включить соответствующий файл:

```
#include <iomanip>
```

Все стандартные параметризованные манипуляторы связаны с форматированием данных, поэтому они будут рассматриваться далее при описании средств форматирования.

Пользовательские манипуляторы

В программе можно определять нестандартные (пользовательские) манипуляторы. Все, что для этого нужно, — написать функцию наподобие приведенной ранее функции `endl()`. Например, следующая функция определяет манипулятор, который игнорирует все символы до конца строки:

```
// io/ignore.hpp
#include <iostream>
#include <limits>

template <class charT, class traits>
inline
std::basic_istream<charT,traits>&
ignoreLine (std::basic_istream<charT,traits>& strm)
{
    // Пропуск символов до конца строки
    strm.ignore(std::numeric_limits<int>::max(),strm.widen('\n'));

    // Возвращение strm для организации цепочечных вызовов
    return strm;
}
```

Манипулятор просто поручает работу функции `ignore()`, которая игнорирует все символы до конца строки (функция `ignore()` описана на с. 584).

Применение манипулятора выглядит очень просто:

```
// Пропустить символы до конца строки
std::cin >> ignoreLine;
```

Многократный вывод манипулятора позволяет проигнорировать сразу несколько строк:

```
// Пропустить две строки
std::cin >> ignoreLine >> ignoreLine;
```

Такая конструкция работает, поскольку вызов функции `ignore(max, c)` означает пропуск всех символов, пока не обнаружится символ `c` во входном потоке данных (или пока не будет прочитано максимальное количество символов, или пока не будет достигнут конец потока данных). Но этот символ также пропускается перед возвращением из функции.



Форматирование

Форматы ввода-вывода в наибольшей степени зависят от двух факторов. Первый и наиболее очевидный — форматные флаги, которые, в частности, определяют точность вывода чисел, символ-заполнитель и основание системы счисления. Другой фактор — настройка форматов под национальные стандарты.

Здесь представлены форматные флаги, а аспекты форматирования, относящиеся к интернационализации программ, описаны на с. 601 и в главе 14.

Форматные флаги

Класс `ios_base` содержит ряд переменных, предназначенных для определения форматов ввода-вывода. В частности, эти переменные определяют минимальную ширину поля, точность вывода вещественных чисел и заполнитель. Переменная типа `ios::fmtflags` содержит флаги конфигурации, которые определяют, нужно ли выводить знак перед положительными числами, должны ли логические значения выводиться в числовом или в символьном виде, и т. д.

Некоторые форматные флаги составляют группы. Например, признаки восьмеричного, десятичного и шестнадцатеричного форматов целых чисел входят в группу. Определены специальные маски, упрощающие работу с группами.

В табл. 13.9 перечислены функции, предназначенные для работы со всеми определениями форматов для потоков данных.

Таблица 13.9. Название таблицы

Функция	Описание
<code>setf(флаги)</code>	Устанавливает флаги, переданные в аргументе, в качестве дополнительных форматных флагов и возвращает предыдущее состояние всех флагов
<code>setf(флаги, маска)</code>	Устанавливает флаги, переданные в первом аргументе, в качестве новых форматных флагов для группы, которая идентифицируется маской, переданной во втором аргументе, и возвращает предыдущее состояние всех флагов
<code>unsetf(флаги)</code>	Сбрасывает флаги, переданные в аргументе
<code>flags()</code>	Возвращает весь набор форматных флагов
<code>flags(флаги)</code>	Устанавливает флаги, переданные в аргументе, в качестве новых форматных флагов и возвращает предыдущее состояние всех флагов
<code>copy(поток)</code>	Копирует все определения форматов из потока, переданного в аргументе

Функции `setf()` и `unsetf()` соответственно устанавливают и сбрасывают один или несколько флагов. Чтобы операция выполнялась сразу с несколькими флагами, следует объединить их оператором `|` (поразрядная дизъюнкция). Во втором аргументе функции `setf()` может передаваться маска, по которой сбрасываются все флаги группы перед установкой флагов, передаваемых в первом аргументе (и также ограниченных группой). Такая возможность не поддерживается версией `setf()`, вызываемой с одним аргументом. Пример:

```
// Установка флагов showpos и uppercase
std::cout.setf (std::ios::showpos | std::ios::uppercase);

// Установка только флага hex в группе basefield
std::cout.setf (std::ios::hex, std::ios::basefield);
```

```
// Сброс флага uppercase  
std::cout.unsetf (std::ios::uppercase);
```

Функция `flags()` позволяет работать со всеми форматными флагами сразу. Вызов `flags()` без аргумента возвращает текущие форматные флаги. При вызове `flags()` с передачей аргумента этот аргумент интерпретируется как новое состояние всех форматных флагов, а функция возвращает старое состояние. Следовательно, при вызове `flags()` с аргументом все флаги сбрасываются, а устанавливаются только переданные флаги. Например, при помощи функции `flags()` можно сохранить текущее состояние флагов для последующего восстановления. В следующем фрагменте показано, как это делается:

```
using std::ios, std::cout;  
  
// Сохранение текущих форматных флагов  
ios::fmtflags oldFlags = cout.flags();  
  
// Изменение состояния флагов  
cout.setf(ios::showpos | ios::showbase | ios::uppercase);  
cout.setf(ios::internal, ios::adjustfield);  
cout << std::hex << x << std::endl;  
  
// Восстановление сохраненных форматных флагов  
cout.flags(oldFlags);
```

Функция `copyfmt()` позволяет скопировать состояние форматных флагов из одного потока данных в другой. Пример приведен на с. 627.

Установка и сброс форматных флагов могут производиться при помощи манипуляторов, представленных в табл. 13.10.

Таблица 13.10. Манипуляторы для работы с форматными флагами

Манипулятор	Описание
<code>setiosflags(флаги)</code>	Устанавливает форматные флаги, переданные в аргументе, путем вызова функции <code>setf(флаги)</code> для потока данных
<code>resetiosflags(маска)</code>	Сбрасывает все флаги группы, определяемой значением маски, путем вызова функции <code>setf(0, маска)</code> для потока данных

Манипуляторы `setiosflags()` и `resetiosflags()` дают возможность соответственно установить или сбросить один или несколько флагов в командах записи или чтения с использованием оператора `<<` или `>>`. Чтобы использовать эти манипуляторы в программе, в нее необходимо включить заголовочный файл `<iomanip>`. Пример:

```
#include <iostream>  
#include <iomanip>  
...  
...
```

```
std::cout << resetiosflags(std::ios::adjustfield) // Сброс выравнивания
      << setiosflags(std::ios::left);           // Левое выравнивание
```

Некоторые операции с флагами выполняются при помощи специализированных манипуляторов. Благодаря своему удобству и наглядности эти манипуляторы используются достаточно часто. Они рассматриваются далее.

Форматированный ввод-вывод логических данных

Флаг `boolalpha` определяет формат ввода и вывода логических значений — числовой или текстовый (табл. 13.11).

Таблица 13.11. Флаг представления логических значений

Флаг	Описание
<code>boolalpha</code>	При установленном флаге используется текстовое представление
	При сброшенном флаге используется числовое представление

Если флаг не установлен (значение по умолчанию), логические данные представляются в числовом виде. В этом случае `false` всегда представляется значением 0, а `true` — значением 1. При чтении логических данных в числовом представлении наличие символов, отличных от 0 и 1, считается ошибкой (для потока данных устанавливается бит `failbit`).

При установке флага логические данные читаются и записываются в текстовом представлении. При чтении логического значения строка должна соответствовать текстовому представлению `true` или `false`. Строки, представляющие эти значения, определяются состоянием объекта локального контекста (см. с. 601 и 670). Стандартный объект локального контекста "С" использует для представления логических значений строки "true" и "false".

Для удобства работы с этим флагом определены специальные манипуляторы (табл. 13.12).

Таблица 13.12. Манипуляторы представления логических данных

Манипулятор	Описание
<code>boolalpha</code>	Включает текстовое представление (установка флага <code>ios::boolalpha</code>)
<code>noboolalpha</code>	Включает числовое представление (сброс флага <code>ios::boolalpha</code>)

Например, следующий фрагмент выводит переменную `b` сначала в числовом, а затем в текстовом представлении:

```
bool b;
...
cout << noboolalpha << b << " == " << boolalpha << b << endl;
```

Ширина поля, заполнитель, выравнивание

Функции `width()` и `fill()` определяют ширину поля и заполнитель (табл. 13.13).

Таблица 13.13. Функции для работы с шириной поля и заполнителем

Функция	Описание
<code>width()</code>	Возвращает текущую ширину поля
<code>width(val)</code>	Задает ширину поля равной <code>val</code> и возвращает предыдущую ширину поля
<code>fill()</code>	Возвращает текущий заполнитель
<code>fill(c)</code>	Назначает новый и возвращает предыдущий заполнитель

Использование при выводе ширины поля, заполнителя и выравнивания

При выводе функция `width()` определяет минимальную ширину поля. Определение относится только к следующему выводимому форматированному полю. При вызове без аргументов `width()` возвращает текущую ширину поля. При вызове с целочисленным аргументом функция `width()` изменяет ширину поля и возвращает ее предыдущее значение. По умолчанию минимальная ширина равна 0; это означает, что размер поля может быть произвольным. Такое значение устанавливается после вывода.

Ширина поля не может использоваться для сокращения вывода. То есть максимальную ширину поля задать невозможно. Вместо этого ее придется самостоятельно запрограммировать, например, записав данные в строку и ограничив вывод определенным количеством символов.

Функция `fill()` определяет символ, используемый для заполнения промежутков между отформатированным представлением величины и позицией, отмечающей минимальную ширину поля. По умолчанию заполнителем является пробел.

Тип выравнивания данных внутри поля определяется тремя флагами, перечисленными в табл. 13.14. Эти флаги определяются в классе `ios_base` вместе с соответствующей маской.

Таблица 13.14. Флаги типа выравнивания

Маска	Флаг	Описание
<code>adjustfield</code>	<code>left</code>	Выравнивание по левому краю
	<code>right</code>	Выравнивание по правому краю
	<code>internal</code>	Выравнивание знака по левому краю, а значения — по правому краю
	нет	Выравнивание по правому краю (по умолчанию)

После выполнения любой операции форматированного ввода-вывода восстанавливается ширина поля по умолчанию. Заполнитель и тип выравнивания остаются без изменений до тех пор, пока они не будут модифицированы явно.

В табл. 13.15 показаны результаты применения разных флагов. В качестве заполнителя используется символ подчеркивания.

Таблица 13.15. Примеры выравнивания

Флаг	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

В ходе стандартизации были изменены правила выравнивания отдельных символов. До стандартизации при выводе отдельных символов ширина поля игнорировалась. Она использовалась при следующей операции форматированного вывода, в которой выводился не одиночный символ. Ошибка была исправлена, но это привело к нарушению совместимости с программами, в которых эта особенность использовалась.

В табл. 13.16 перечислены манипуляторы, изменяющие ширину поля, заполнитель и тип выравнивания.

Таблица 13.16. Манипуляторы для управления шириной поля, заполнителем и типом выравнивания

Манипулятор	Описание
setw(val)	Устанавливает ширину поля при вводе и выводе равной val (соответствует функции width())
setfill(c)	Назначает заполнителем символ c (соответствует функции fill())
left	Устанавливает выравнивание по левому краю
right	Устанавливает выравнивание по правому краю
internal	Устанавливает выравнивание знака по левому краю, а значения — по правому краю

Манипуляторам setw() и setfill() должен передаваться аргумент, поэтому для их использования в программу необходимо включить заголовочный файл <iomanip>. Например, рассмотрим такой фрагмент:

```
#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;
```

Этот фрагмент выводит следующий результат:

```
_3.14 42
sum: _ _ _ _ _ _ _ _ 42
```

Использование при вводе ширины поля

Ширина поля также позволяет задать максимальное количество символов, вводимых при чтении последовательностей символов типа `char*`. Если значение `width()` отлично от 0, то из потока данных читаются не более `width()-1` символов.

Поскольку обычные С-строки не могут увеличиваться при чтении данных, при их чтении оператором `>>` всегда следует ограничивать максимальный размер ввода функциями `width()` или `setw()`. Пример:

```
char buffer[81];
```

```
// Чтение не более 80 символов
cin >> setw(sizeof(buffer)) >> buffer;
```

Функция читает не более 80 символов, хотя `sizeof(buffer)` возвращает 81, поскольку один символ является признаком завершения строки (он присоединяется автоматически). Обратите внимание на распространенную ошибку:

```
char* s;
cin >> setw(sizeof(s)) >> s; // ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ
```

Дело в том, что `s` объявляется как указатель без выделения памяти под символы, поэтому `sizeof(s)` возвращает размер указателя вместо размера памяти, на которую он ссылается. Это одна из типичных ошибок, возникающих при работе с С-строками. Строковые классы позволяют предотвратить подобные ошибки:

```
string buffer;
cin >> buffer; // OK .
```

Отображение знака для положительных чисел вывод в верхнем регистре

В табл. 13.17 представлены форматные флаги `showpos` и `uppercase`, определяющие общий вид числовых значений.

Таблица 13.17. Флаги управления знаком и регистром символов в числовых значениях

Манипулятор	Описание
<code>showpos</code>	Вывод знака для положительных чисел
<code>uppercase</code>	Вывод символов в верхнем регистре

Установка флага `ios::showpos` означает, что положительные числа должны выводиться со знаком. Если флаг сброшен, то со знаком выводятся только отрицательные числа. Флаг `ios::uppercase` означает, что буквы в числовых значениях должны выводиться в верхнем регистре. Этот флаг распространяется как на целые числа, записанные в шестнадцатеричном виде, так и на вещественные числа в научной (экспоненциальной) записи. По умолчанию положительные числа

выводятся без знака, а символы выводятся в нижнем регистре. Например, рассмотрим такой фрагмент:

```
std::cout << 12345678.9 << std::endl;
std::cout.setf (std::ios::showpos | std::ios::uppercase);
std::cout << 12345678.9 << std::endl;
```

Этот фрагмент выводит следующий результат:

```
1.23457e+07
+1.23457E+07
```

Оба флага также можно устанавливать и сбрасывать при помощи манипуляторов, представленных в табл. 13.18.

Таблица 13.18. Манипуляторы для управления знаком и регистром символов в числах

Манипулятор	Описание
showpos	Вывод знака для положительных чисел (установка флага ios::showpos)
noshowpos	Вывод положительных чисел без знака (сброс флага ios::showpos)
uppercase	Вывод символов в числах в верхнем регистре (установка флага ios::uppercase)
nouppercase	Вывод символов в числах в нижнем регистре (установка флага ios::uppercase)

Система счисления

Следующая группа из трех флагов управляет основанием системы счисления, используемой при вводе-выводе целых чисел (табл. 13.19). Флаги определяются в классе `ios_base` вместе с соответствующей маской.

Таблица 13.19. Флаги системы счисления

Маска	Флаг	Описание
basefield	oct	Чтение/запись в восьмеричной системе
	dec	Чтение/запись в десятичной системе (используется по умолчанию)
	hex	Чтение/запись в шестнадцатеричной системе
	нет	Запись в шестнадцатеричной системе, чтение в зависимости от начальных символов целого числа

Смена основания системы счисления отражается на дальнейшем вводе-выводе всех целых чисел до следующего изменения флагов. По умолчанию используется десятичный формат. Поддержка двоичной записи не предусмотрена, однако чтение и запись целых чисел в двоичном виде может осуществляться при помощи класса `bitset`. За дополнительной информацией обращайтесь к с. 445.

Флаги системы счисления также распространяются на ввод. Данные читаются в системе, определяемой установкой одного из флагов. Если флаги не установлены, то при чтении основание системы счисления определяется по префиксам:

число, начинающееся с префикса `0x` или `0X`, интерпретируется как шестнадцатеричное. Префикс `0` является признаком восьмеричной записи. Во всех остальных случаях число считается десятичным.

- Существуют два основных способа переключения флагов системы счисления.
- Сброс одного флага и установка другого:

```
std::cout.unsetf (std::ios::dec);
std::cout.setf (std::ios::hex);
```

- Установка одного флага с автоматическим сбросом остальных флагов группы:

```
std::cout.setf (std::ios::hex, std::ios::basefield);
```

В табл. 13.20 перечислены манипуляторы, упрощающие работу с флагами этой группы.

Таблица 13.20. Манипуляторы, определяющие основание системы счисления

Манипулятор	Описание
<code>oct</code>	Запись и чтение в восьмеричной системе
<code>dec</code>	Запись и чтение в десятичной системе
<code>hex</code>	Запись и чтение в шестнадцатеричной системе

Например, следующий фрагмент выводит `x` и `y` в шестнадцатеричной системе, а `z` — в десятичной системе:

```
int x, y, z;
...
std::cout << std::ios::hex << x << std::endl;
std::cout << y << ' ' << std::ios::dec << z << std::endl;
```

Дополнительный флаг `showbase` выводит числа по стандартным правилам обозначения системы счисления числовых литералов в C/C++ (табл. 13.21).

Таблица 13.21. Флаг, обозначающий основание системы счисления

Флаг	Описание
<code>showbase</code>	Если флаг установлен, при выводе указывается система счисления числовых литералов

При установке флага `ios::showbase` восьмеричные числа выводятся с префиксом `0`, а шестнадцатеричные числа — с префиксом `0x` (или при установленном флаге `ios::uppercase` — `0X`). Например, рассмотрим такой фрагмент:

```
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;
```

Этот фрагмент выводит следующий результат:

```
127 255
7f ff
0x7f 0xff
0X7F 0xFF
```

Для управления флагом `ios::showbase` также можно использовать манипуляторы, представленные в табл. 13.22.

Таблица 13.22. Манипуляторы для идентификации системы счисления

Манипулятор	Описание
<code>showbase</code>	Вывод идентификатора системы счисления (установка флага <code>ios::showbase</code>)
<code>noshowbase</code>	Запрет на вывод идентификатора системы счисления (сброс флага <code>ios::showbase</code>)

Формат вещественных чисел

Некоторые флаги и переменные управляют выводом вещественных чисел. Флаги, перечисленные в табл. 13.23, определяют тип записи (десятичная или научная). Эти флаги определяются в классе `ios_base` вместе с соответствующей маской. Если флаг `ios::fixed` установлен, вещественные числа выводятся в десятичной записи. При установке флага `ios::scientific` используется научная (экспоненциальная) запись.

Таблица 13.23. Флаги вывода вещественных чисел

Маска	Флаг	Описание
<code>floatfield</code>	<code>fixed</code>	Использование десятичной записи
	<code>scientific</code>	Использование научной записи
	нет	Использование «лучшей» из этих двух записей

Точность вывода определяется функцией `precision()` (табл. 13.24).

Таблица 13.24. Функции управления точностью вывода вещественных чисел

Функция	Описание
<code>precision()</code>	Возвращает текущую точность вывода вещественных чисел
<code>precision(val)</code>	Определяет новую точность вывода вещественных чисел <code>val</code> и возвращает прежнюю точность

При использовании научной записи функция `precision()` определяет количество десятичных разрядов в дробной части. Остаток всегда округляется.

Вызов `precision()` без аргументов возвращает текущую точность. При вызове с аргументом функция `precision()` устанавливает заданную точность вывода и возвращает предыдущую точность. По умолчанию точность равна шести десятичным цифрам.

По умолчанию ни один из флагов `ios::fixed` и `ios::scientific` не устанавливается. В этом случае запись выбирается в зависимости от выводимого значения. Для этого делается попытка вывести все значащие десятичные цифры (но не более `precision()`) с удалением начального нуля перед десятичной точкой и/или всех завершающих пробелов, а в крайнем случае — даже десятичной точки. Если `precision()` разрядов оказывается достаточно, используется десятичная запись; в противном случае — научная запись.

При помощи флага `showpoint` можно заставить поток данных выводить десятичную точку и завершающие нули до ширины `precision()` разрядов (табл. 13.25).

Таблица 13.25. Флаг обязательного вывода десятичной точки

Флаг	Описание
<code>showpoint</code>	Десятичная точка всегда используется при выводе

Таблица 13.26 иллюстрирует довольно сложные взаимосвязи между флагами и точностью на примере двух конкретных чисел.

Таблица 13.26. Примеры форматирования вещественных чисел

	<code>precision()</code>	421.0	0.0123456789
Обычный формат	2	4.2e+02	0.012
	6	421	0.0123457
С флагом <code>showpoint</code>	2	4.2e+02	0.012
	6	421.000	0.0123457
С флагом <code>fixed</code>	2	421.00	0.01
	6	421.000000	0.012346
С флагом <code>scientific</code>	2	4.21e+02	1.23e-02
	6	4.210000e+02	1.234568e-02

Как и в случае целых значений, флаг `ios::showpos` служит для принудительного вывода знака положительных чисел. Флаг `ios::uppercase` указывает, какая буква должна использоваться в научной записи (`E` или `e`).

Флаг `ios::showpoint`, тип записи и точность можно задать при помощи манипуляторов, представленных в табл. 13.27.

Например, рассмотрим такую команду:

```
std::cout << std::scientific << std::showpoint
    << std::setprecision(8)
    << 0.123456789 << std::endl;
```

Таблица 13.27. Манипуляторы формата вывода вещественных чисел

Манипулятор	Описание
showpoint	Десятичная точка всегда используется при выводе (установка флага <code>ios::showpoint</code>)
noshowpoint	Десятичная точка не обязательна при выводе (сброс флага <code>ios::showpoint</code>)
setprecision(val)	Выбор новой точности <code>val</code>
fixed	Использование десятичной записи.
scientific	Использование научной записи

Эта команда выводит следующий результат:

1.23456789e-01

Манипулятору `setprecision()` передается аргумент, поэтому для его использования необходимо включить в программу заголовочный файл `<iomanip>`.

Общие параметры форматирования

Список форматных флагов завершается флагами `skipws` и `unitbuf` (табл. 13.28).

Таблица 13.28. Оставшиеся форматные флаги

Флаг	Описание
<code>skipws</code>	Автоматическое игнорирование начальных пропусков при чтении данных оператором <code>>></code>
<code>unitbuf</code>	Принудительный вывод содержимого буфера после каждой операции записи

Флаг `ios::skipws` устанавливается по умолчанию; это означает, что по умолчанию некоторые операции чтения игнорируют начальные пропуски. Обычно этот флаг удобнее держать установленным. Например, вам не придется специально заботиться о чтении пробелов, разделяющих числа. С другой стороны, это означает, что вы не сможете читать пробелы оператором `>>`, потому что начальные пропуски всегда игнорируются.

Флаг `ios::unitbuf` управляет буферизацией вывода. При установленном флаге `ios::unitbuf` вывод практически выполняется без буферизации — выходной буфер очищается после каждой операции записи. По умолчанию этот флаг не устанавливается. Исключение составляют потоки данных `cerr` и `w cerr`, для которых этот флаг устанавливается в исходном состоянии.

В табл. 13.29 представлены манипуляторы, используемые для управления дополнительными флагами.

Таблица 13.29. Манипуляторы для управления дополнительными форматными флагами

Манипулятор	Описание
<code>skipws</code>	Автоматическое игнорирование начальных пропусков при чтении данных оператором <code>>></code> (установка флага <code>ios::skipws</code>)

Манипулятор	Описание
noskipws	Обработка начальных пропусков при чтении данных оператором >> (сброс флага ios::skipws)
unitbuf	Принудительный вывод содержимого буфера после каждой операции записи (установка флага ios::unitbuf)
nounitbuf	Отмена принудительного вывода содержимого буфера после каждой операции записи (установка флага ios::unitbuf)

Интернационализация

Форматы ввода-вывода также адаптируются к национальным стандартам. Функции, определенные для этой цели в классе `ios_base`, перечислены в табл. 13.30.

Таблица 13.30. Функции интернационализации

Функция	Описание
<code>imbue(loc)</code>	Назначение объекта локального контекста
<code>getloc()</code>	Получение текущего объекта локального контекста

С каждым потоком данных связывается некоторый объект локального контекста. По умолчанию исходный объект локального контекста создается как копия глобального объекта локального контекста на момент конструирования потока данных. В частности, объект локального контекста определяет параметры форматирования чисел (например, символ, используемый в качестве десятичной точки, или режим числового/строкового представления логических величин).

В отличие от аналогичных средств С средства интернационализации стандартной библиотеки C++ позволяют задавать локальные контексты на уровне отдельных потоков данных. Например, такая возможность позволяет выполнять чтение вещественных чисел в американском формате и последующей записи в немецком формате (в котором вместо «десятичной точки» используется запятая). На с. 665 представлен подробный пример.

При работе с потоками данных часто возникает задача приведения отдельных символов (в первую очередь управляющих) к кодировке потока данных. Для этого в потоках данных поддерживаются функции преобразования, представленные в табл. 13.31.

Таблица 13.31. Вспомогательные функции интернационализации

Функция	Описание
<code>widen(c)</code>	Преобразование символа с типа <code>char</code> к кодировке, используемой потоком
<code>narrow(c,def)</code>	Преобразование символа с из кодировки, используемой потоком, к типу <code>char</code> (если такого символа не существует, возвращается <code>def</code>)

Например, следующая команда преобразует символ новой строки в кодировку, используемую потоком данных:

```
strm.widen('\n')
```

За дополнительной информацией о локальных контекстах и интернационализации обращайтесь к главе 14.

Доступ к файлам

Потоки данных также используются для работы с файлами. В стандартную библиотеку C++ входят четыре основных шаблона, для которых определены стандартные специализации.

- Шаблон `basic_ifstream<>` со специализациями `ifstream` и `wifstream` обеспечивает чтение файлов («файловый входной поток данных»).
- Шаблон `basic_ofstream<>` со специализациями `ofstream` и `wofstream` обеспечивает запись файлов («файловый выходной поток данных»).
- Шаблон `basic_fstream<>` со специализациями `fstream` и `wfstream` обеспечивает чтение и запись файлов.
- Шаблон `basic_filebuf<>` со специализациями `filebuf` и `wfilebuf` используется только другими классами файловых потоков данных для выполнения фактических операций чтения и записи символов.

Иерархия классов файловых потоков данных, представленная на рис. 13.2, соответствует общей иерархии базовых классов потоков данных.

Эти классы определяются в заголовочном файле `<fstream>` следующим образом:

```
namespace std {
    template <class charT,
              class traits = char_traits<charT> >
        class basic_ifstream;
    typedef basic_ifstream<char> ifstream;
    typedef basic_ifstream<wchar_t> wifstream;

    template <class charT,
              class traits = char_traits<charT> >
        class basic_ofstream;
    typedef basic_ofstream<char> ofstream;
    typedef basic_ofstream<wchar_t> wofstream;

    template <class charT,
              class traits = char_traits<charT> >
        class basic_fstream;
    typedef basic_fstream<char> fstream;
    typedef basic_fstream<wchar_t> wfstream;

    template <class charT,
              class traits = char_traits<charT> >
        class basic_filebuf;
}
```

```

class basic_filebuf;
typedef basic_filebuf<char>    filebuf;
typedef basic_filebuf<wchar_t> wfilebuf;
}

```

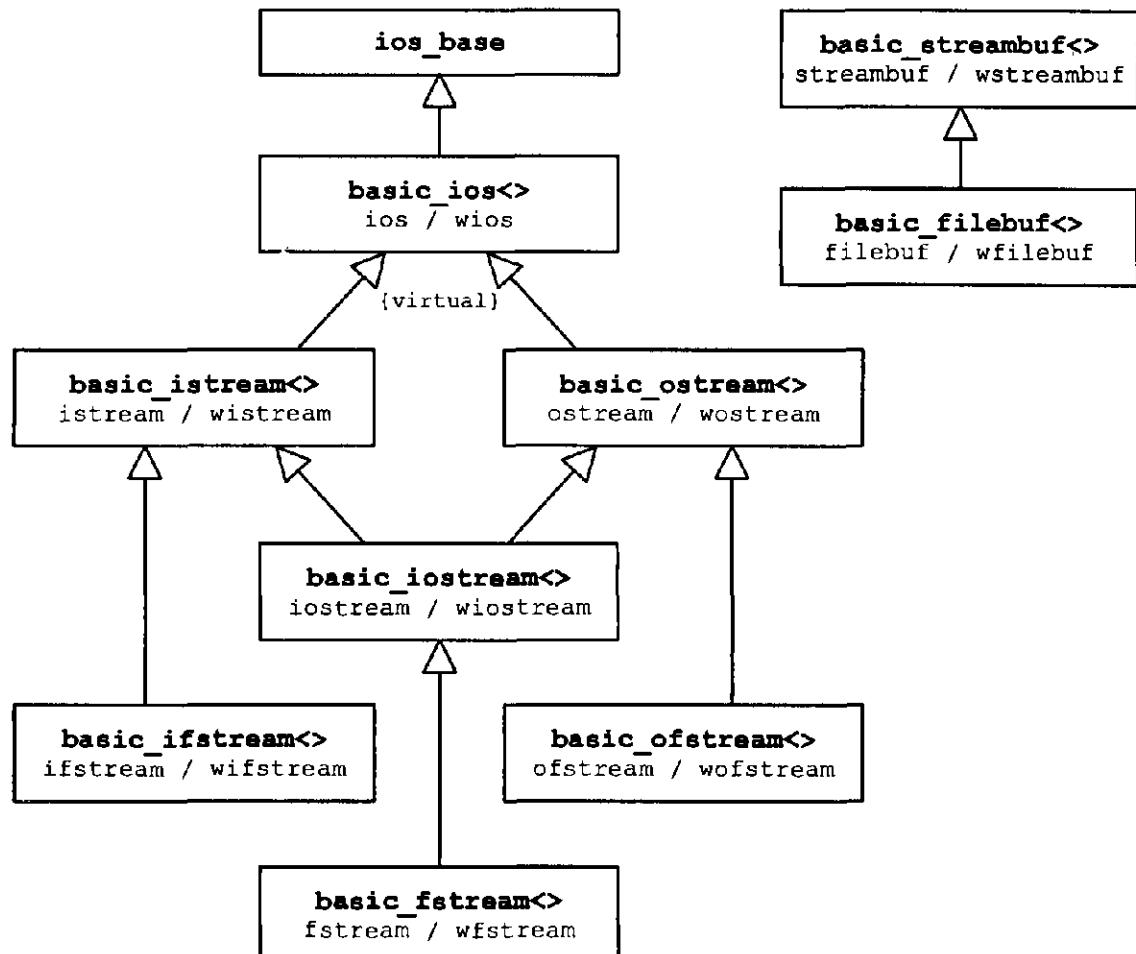


Рис. 13.2. Иерархия классов файловых потоков данных

Основным достоинством потоковых классов для работы с файлами является автоматизация выполняемых операций. Файлы автоматически открываются во время конструирования и закрываются при уничтожении объекта. Естественно, что эта возможность имеется благодаря соответствующему определению конструкторов и деструкторов.

Одно важное обстоятельство, относящееся к потокам данных с поддержкой и чтения и записи, — такие потоки *не должны* допускать произвольного переключения между чтением и записью¹! Чтобы после начала чтения из файла переключиться на запись (или наоборот), необходимо выполнить операцию позиционирования (возможно, с сохранением текущей позиции). Единственное исключение из этого правила относится к чтению с выходом за конец файла;

¹ Это ограничение было унаследовано от С. Тем не менее оно с большой вероятностью будет поддерживаться и будущими реализациями стандартной библиотеки С++.

в этой ситуации можно немедленно переходить к записи символов. Нарушение этого ограничения приводит к нежелательным побочным эффектам.

Если при конструировании файлового потока данных в аргументе передается С-строка (тип `char*`), то при этом автоматически делается попытка открыть файл для чтения и/или записи. Признак успеха этой попытки отражается в состоянии потока данных. Следовательно, после конструирования следует проверить состояние потока данных.

Следующая программа открывает файл `charset.out` и записывает в него текущий набор символов (все символы в интервале 32–255):

```
// io/charset.cpp
#include <string>           // Строки
#include <iostream>          // Ввод-вывод
#include <fstream>           // Файловый ввод-вывод
#include <iomanip>           // setw()
#include <cstdlib>           // exit()
using namespace std;

// Определяющие объявления
void writeCharsetToFile (const string& filename);
void outputFile (const string& filename);

int main ()
{
    writeCharsetToFile("charset.out");
    outputFile("charset.out");
}

void writeCharsetToFile (const string& filename)
{
    // Открытие выходного файла
    ofstream file(filename.c_str());

    // Файл открыт?
    if (!file) {
        // NO. abort program
        cerr << "can't open output file \""
            << filename << "\""
            << endl;
        exit(EXIT_FAILURE);
    }

    // Вывод текущего набора символов
    for (int i=32; i<256; i++) {
        file << "value: " << setw(3) << i << " "
            << "char: " << static_cast<char>(i) << endl;
    }
} // Автоматическое закрытие файла

void outputFile (const string& filename)
```

```
{  
    // open input file  
    ifstream file(filename.c_str());  
  
    // Файл открыт?  
    if (!file) {  
        // НЕТ, аварийное завершение программы  
        cerr << "can't open input file \"\" << filename << "\""  
            << endl;  
        exit(EXIT_FAILURE);  
    }  
  
    // Копирование содержимого файла в cout  
    char c:  
    while (file.get(c)) {  
        cout.put(c);  
    }  
}  
// Автоматическое закрытие файла
```

В функции `writeCharsetToFile()` конструктор класса `ofstream` открывает файл с заданным именем:

```
std::ofstream file(filename.c_str());
```

Имя файла передается в типе `string`, поэтому его преобразование в `const char*` выполняется функцией `c_str()` (см. с. 467). К сожалению, у классов файловых потоков данных не существует конструктора, вызываемого с аргументом типа `string`. После преобразования программа проверяет состояние потока данных:

```
if (!file) {  
    ...  
}
```

Если открыть поток данных не удалось, условие будет истинным. После проверки программа в цикле выводит числа от 32 до 255 вместе с соответствующими символами.

Внутри функции `outputFile()` файл открывается конструктором класса `ifstream`, после чего происходит посимвольная запись нового содержимого файла.

В конце обеих функций открытые файлы автоматически закрываются при выходе соответствующих потоков данных из области видимости. Деструкторы классов `Ifstream` и `ofstream` закрывают файлы, если они остаются открытыми на момент уничтожения объекта.

Если файл должен использоваться за пределами области видимости, в которой он был создан, выделите объект из кучи и удалите его позднее, когда надобность в нем отпадет:

```
std::ofstream* filePtr = new std::ofstream("xyz");  
...  
delete filePtr;
```

В таких случаях следует использовать классы умных указателей, например `CountedPtr` (см. с. 226) или `auto_ptr` (см. с. 54).

Вместо последовательного вывода отдельных символов также можно вывести все содержимое файла одной командой, передавая указатель на потоковый буфер файла в аргументе оператора `<<`:

```
std::cout << file.rdbuf();
```

Подробности приведены на с. 656.

Режимы открытия файлов

В табл. 13.32 перечислены флаги управления режимами открытия файлов, определенные в классе `ios_base`. Флаги относятся к типу `openmode` и группируются в битовые маски по аналогии с флагами `fmtflags`.

Таблица 13.32. Флаги открытия файлов

Флаг	Описание
<code>in</code>	Открытие файла для чтения (используется по умолчанию для <code>ifstream</code>)
<code>out</code>	Открытие файла для записи (используется по умолчанию для <code>ofstream</code>)
<code>app</code>	Запись данных производится только в конец файла
<code>ate</code>	Позиционирование в конец файла после открытия («at end»)
<code>trunc</code>	Удаление старого содержимого файла
<code>binary</code>	Специальные символы не заменяются

Флаг `binary` запрещает преобразование специальных символов или символьных последовательностей (например, конца строки или конца файла). В операционных системах типа MS-DOS или OS/2 конец логической строки в тексте обозначается двумя символами (CR и LF). При открытии файла в обычном текстовом режиме (брошенный флаг `binary`) символы новой строки заменяются последовательностью из двух символов, и наоборот. При открытии файла в двоичном режиме (с установленным флагом `binary`) эти преобразования не выполняются.

Флаг `binary` должен использоваться всегда, когда файл не содержит чисто текстовой информации и обрабатывается как двоичные данные. Пример — копирование файла с последовательным чтением символов и их записью без модификации. Если файл обрабатывается в текстовом виде, флаг `binary` не устанавливается, потому что в этом случае символы новой строки нуждаются в специальной обработке.

В некоторых реализациях имеются дополнительные флаги типа `poscreate` (файл должен существовать при открытии) и `noreplace` (файл не должен существовать). Однако эти флаги отсутствуют в стандарте, поэтому их использование влияет на переносимость программы.

Флаги объединяются оператором `|`. Полученный результат типа `openmode` может передаваться конструктору во втором аргументе. Например, следующая команда открывает файл для присоединения текста в конце:

```
std::ofstream file("xyz.out", std::ios::out|std::ios::app);
```

В табл. 13.33 представлены различные комбинации флагов и их аналоги — строковые обозначения режимов, используемые функцией открытия файлов `fopen()` в интерфейсе языка С. Комбинации с флагами `binary` и `ate` не приводятся. Установленный флаг `binary` соответствует строке с присоединенным символом `b`, а установленный флаг `ate` соответствует позиционированию в конец файла немедленно после открытия. Другие комбинации, отсутствующие в таблице (например, `trunc|app`), недопустимы.

Таблица 13.33. Описание режимов открытия файлов в C++

Флаги <code>ios_base</code>	Описание	Обозначения режимов в С
<code>in</code>	Чтение (файл должен существовать)	"r"
<code>out</code>	Стирание и запись (файл создается при необходимости)	"w"
<code>out trunc</code>	Стирание и запись (файл создается при необходимости)	"w"
<code>out app</code>	Присоединение (файл создается при необходимости)	"a"
<code>in out</code>	Чтение и запись с исходным позиционированием в начало файла	"r+"
<code>in out trunc</code>	Стирание, чтение и запись (файл создается при необходимости)	"w+"

Открытие файла для чтения и/или записи не зависит от класса соответствующего объекта потока данных. Класс лишь определяет режим открытия по умолчанию при отсутствии второго аргумента. Это означает, что файлы, используемые только классом `ifstream` или `ofstream`, могут открываться для чтения и записи. Режим открытия передается соответствующему классу потокового буфера, который открывает файл. Тем не менее операции, разрешенные для данного объекта, определяются классом потока данных.

Также существуют три функции для открытия и закрытия файлов, принадлежащих файловым потокам данных (табл. 13.34).

Таблица 13.34. Функции открытия и закрытия файлов

Функция	Описание
<code>open(имя)</code>	Открытие файла для потока в режиме по умолчанию
<code>open(имя, флаги)</code>	Открытие файла для потока в режиме, определяемом переданными флагами
<code>close()</code>	Закрытие файлового потока
<code>Is_open()</code>	Проверка открытия файла

Эти функции используются в основном при создании файловых потоков данных без инициализации. В следующем примере открываются все файлы, имена которых передаются в аргументах, и выводится их содержимое (аналог утилиты `cat` системы UNIX).

```
// io/cat1.cpp
// Заголовочные файлы для файлового ввода-вывода
#include <fstream>
#include <iostream>
using namespace std;

/* Для всех файлов, имена которых переданы в аргументах командной строки.
 * - открыть, вывести содержимое и закрыть файл
 */
int main (int argc, char* argv[])
{
    ifstream file;

    // Перебор аргументов командной строки
    for (int i=1; i<argc; ++i) {

        // Открытие файла
        file.open(argv[i]);

        // Вывод содержимого файла в cout
        char c;
        while (file.get(c)) {
            cout.put(c);
        }

        // Сброс флагов eofbit и failbit, установленных
        // при обнаружении конца файла
        file.clear();

        // Закрытие файла
        file.close();
    }
}
```

Обратите внимание: после завершения обработки файла вызывается функция `clear()` для сброса флагов состояния, установленных при обнаружении конца файла. Этот вызов необходим, поскольку потоковый объект используется для нескольких файлов. Функция `open()` *никогда* не сбрасывает флаги состояния. Следовательно, если поток данных не находится в нормальном состоянии, после его закрытия и повторного открытия вам все равно придется вызвать `clear()`, чтобы сбросить установленные флаги. То же самое следует сделать и при открытии другого файла.

Вместо обработки отдельных символов также можно вывести все содержимое файла одной командой, для чего оператору << в качестве аргумента передается указатель на потоковый буфер файла:

```
// Запись содержимого файла в cout
std::cout << file.rdbuf();
```

Подробности приведены на с. 656.

Тривиальный доступ к файлам

В табл. 13.35 перечислены функции позиционирования в потоках данных C++.

Таблица 13.35. Функции позиционирования в потоках данных

Класс	Функция	Описание
basic_istream<>	tellg()	Возвращает текущую позицию чтения
	seekg(pos)	Устанавливает абсолютную позицию чтения
	seekg(offset, gpos)	Устанавливает относительную позицию чтения
basic_ostream<>	tellp()	Возвращает текущую позицию записи
	seekp(pos)	Устанавливает абсолютную позицию записи
	seekp(offset, gpos)	Устанавливает относительную позицию записи

Позиционирование чтения и записи выполняется отдельными функциями (суффикс «g» означает «get», а суффикс «p» — «put»). Функции чтения определяются в классе **basic_istream**, а функции записи — в классе **basic_ostream**. Тем не менее не все потоковые классы поддерживают позиционирование. Например, для потоков данных **cin**, **cout** и **cerr** позиционирование не определено. Операции файлового позиционирования определяются в базовых классах, потому что обычно используются ссылки на объекты типов **istream** и **ostream**.

Функции **seekg()** и **seekp()** могут вызываться для абсолютных или относительных позиций. Функции **tellg()** и **tellp()** возвращают абсолютную позицию в виде значения типа **pos_type**. Это значение *не является* целым числом или индексом, задающим позицию символа, поскольку логическая позиция может отличаться от фактической. Например, в текстовых файлах MS-DOS символы новой строки хранятся в файлах в виде двух символов, хотя логически они соответствуют только одному символу. Кроме того, ситуация дополнительно усложняется при многобайтовой кодировке символов.

Разобраться в точном определении типа **pos_type** непросто: стандартная библиотека C++ определяет глобальный класс шаблона **fpos<>** для представления позиций в файлах. На базе класса **fpos<>** определяются типы **streampos** (для потоков данных **char**) и **wstreampos** (для потоков данных **wchar_t**). Эти типы используются для определения **pos_type** соответствующих классов трактовок (см. с. 659). Наконец, переменная типа **pos_type** класса трактовок требуется для определения типа **pos_type** соответствующих потоковых классов. Следовательно,

позиции в потоке данных также могут представляться типом `streampos`, но использовать типы `long` и `unsigned long` было бы неправильно, потому что `streampos` не является целочисленным типом (а точнее, перестал им быть¹). Пример:

```
// Сохранение текущей позиции
std::ios::pos_type pos = file.tellg();

...
// Переход к позиции, хранящейся в pos
file.seekg(pos);
```

Следующие объявления эквивалентны:

```
std::ios::pos_type pos;
std::streampos pos;
```

В версиях с относительным позиционированием смещение задается по отношению к трем позициям, для которых определены соответствующие константы (табл. 13.36). Константы определяются в классе `ios_base` и относятся к типу `seekdir`.

Таблица 13.36. Константы относительных позиций

Константа	Описание
<code>beg</code>	Смещение задается относительно начала файла
<code>cur</code>	Смещение задается относительно текущей позиции
<code>end</code>	Смещение задается относительно конца файла

Смещение относится к типу `off_type`, который представляет собой косвенное определение для `streamoff`. По аналогии с `pos_type` тип `streamoff` используется для определения `off_type` в классе трактовок (см. с. 660) и в потоковых классах. Тем не менее `streamoff` является целым знаковым типом, поэтому смещение в потоке данных может задаваться целым числом. Пример:

```
// Позиционирование в начало файла
file.seekg(0, std::ios::beg);

...
// Позиционирование на 20 символов вперед
file.seekg(20, std::ios::cur);

...
// Позиционирование на 10 символов от конца файла
file.seekg(-10, std::ios::end);
```

При позиционировании необходимо всегда следить за тем, чтобы позиция оставалась внутри файла. Позиционирование перед началом или после конца файла приводит к непредсказуемым последствиям.

¹ Ранее тип `streampos` использовался для определения позиций в потоках данных и определялся просто как `unsigned long`.

Следующий пример демонстрирует использование функции `seekg()`. В нем определена функция, которая дважды выводит содержимое файла:

```
// Заголовочные файлы для ввода-вывода
#include <iostream>
#include <fstream>

void printFileTwice (const char* filename)
{
    // Открытие файла
    std::ifstream file(filename);

    // Первый вывод содержимого
    std::cout << file.rdbuf();

    // Возврат к началу файла
    file.seekg(0);

    // Второй вывод содержимого
    std::cout << file.rdbuf();
}

int main (int argc, char* argv[])
{
    // Двукратный вывод всех файлов, переданных в командной строке
    for (int i=1; i<argc; ++i) {
        printFileTwice(argv[i]);
    }
}
```

Обратите внимание на вывод содержимого файла функцией `file.rdbuf()` (см. с. 656). Операция выполняется прямо с потоковым буфером и не изменяет состояния потока данных. Если содержимое `file` выводится функциями потокового интерфейса, такими как `getline()` (см. с. 583), вам придется сбросить состояние файла `file` функцией `clear()` перед тем, как выполнять с ним любые операции, включая изменение позиции чтения, поскольку эти функции устанавливают флаги `ios::eofbit` и `ios::failbit` при достижении конца файла.

Управление позициями чтения и записи осуществляется разными функциями, но для стандартных потоков данных поддерживается общая позиция чтения/записи в одном потоковом буфере. Это важно, если буфер используется несколькими потоками данных. За дополнительной информацией обращайтесь к с. 613.

Файловые дескрипторы

Некоторые реализации позволяют присоединить поток данных к ранее открытому каналу ввода-вывода. Для этого файловый поток данных инициализируется *файловым дескриптором*.

Файловый дескриптор представляет собой целое число, идентифицирующее открытый канал ввода-вывода. В системах семейства UNIX файловые дескрип-

торы используются в низкоуровневом интерфейсе с функциями ввода-вывода операционной системы. Определены три стандартных файловых дескриптора:

- 0 – стандартный канал ввода;
- 1 – стандартный канал вывода;
- 2 – стандартный канал вывода ошибок.

Каналы могут связываться с файлами, консолью, процессами или другими средствами ввода-вывода.

К сожалению, стандартная библиотека C++ не поддерживает присоединение потоков данных к каналам ввода-вывода при помощи файловых дескрипторов. Это объясняется тем, что проектировщики стремились обеспечить независимость от конкретных особенностей операционных систем. Впрочем, на практике такая возможность существует, а ее единственный недостаток — влияние на переносимость программ. На сегодняшний день в стандартах интерфейсов операционных систем (таких, как POSIX или X/OPEN) не существует такой спецификации, причем ее разработка даже не планируется.

И все же поток данных можно инициализировать по файловому дескриптору. Описание и реализация возможного решения приводятся на с. 641.

Связывание потоков ввода-вывода

Иногда перед программистом возникает задача связывания двух потоков данных. Предположим, вы хотите, чтобы перед вводом данных на экран выводился текст с запросом на ввод. Другой пример — чтение и запись в одном потоке данных (это относится в основном к файлам). Третий характерный пример — выполнение операций с потоком данных в разных форматах. Все эти примеры будут рассмотрены далее.

Нежесткое связывание функцией `tie`

При связывании потока с выходным потоком данных их буферы синхронизируются так, что содержимое буфера выходного потока данных будет очищаться функцией `flush()` перед любой операцией ввода или вывода в другом потоке данных. В табл. 13.37 перечислены функции связывания потоков данных, определенные в классе `basic_ios`.

Таблица 13.37. Связывание потоков данных

Функция	Описание
<code>tie()</code>	Возвращает указатель на выходной поток, связанный с данным потоком
<code>tie(ostream* strm)</code>	Связывает поток с входным потоком, заданным аргументом, и возвращает указатель на предыдущий связанный входной поток (если он был)

Вызов функции `tie()` без аргументов возвращает указатель на текущий выходной поток данных, связанный с данным потоком. Чтобы связать поток данных

с новым выходным потоком, следует передать указатель на этот выходной поток в аргументе `tie()`. Аргумент является указателем, поскольку в нем также может передаваться 0 (или `NULL`) для разрыва связи с выходным потоком данных. Каждый поток данных в любой момент времени может быть связан только с одним выходным потоком, но один выходной поток может быть связан с несколькими потоками данных.

По умолчанию этот механизм используется для связывания стандартного входного потока данных со стандартным выходным потоком:

```
// Стандартные связи:  
std::cin.tie(&std::cout);  
std::wcin.tie(&std::wcout);
```

Тем самым гарантируется, что сообщение с запросом на ввод будет выведено из буфера перед операцией ввода. Например, при выполнении следующего фрагмента перед чтением `x` для `cout` будет автоматически вызвана функция `flush()`:

```
std::cout << "Please enter x: ";  
std::cin >> x;
```

Чтобы разорвать связь между потоками данных, передайте 0 или `NULL` при вызове `tie()`. Пример:

```
// Отсоединение cin от выходных потоков  
std::cin::tie(static_cast<std::ostream*>(0));
```

Это может ускорить работу программы, поскольку позволяет избежать лишней очистки потоковых буферов (проблемы эффективности работы с потоками данных рассматриваются на с. 654).

Выходной поток данных также можно связать с другим выходным потоком. Например, следующая команда указывает, что перед выводом каких-либо данных в `cerr` необходимо очистить буфер стандартного выходного потока данных:

```
// Связывание cout с cerr  
cerr.tie(&cout);
```

Жесткое связывание с использованием потоковых буферов

Функция `rdbuf()` осуществляет жесткое связывание потоков данных с помощью общего потокового буфера (табл. 13.38).

Таблица 13.38. Работа с потоковыми буферами

Функция	Описание
<code>rdbuf()</code>	Возвращает указатель на потоковый буфер
<code>rdbuf(streambuf*)</code>	Назначает потоковый буфер, определяемый аргументом, и возвращает указатель на предыдущий потоковый буфер

Функция `rdbuf()` позволяет нескольким объектам потоков данных читать из одного входного канала или записывать в один выходной канал без нарушения порядка ввода-вывода. Однако использование нескольких потоковых буферов создает проблемы из-за буферизации операций ввода-вывода. Следовательно, применение для одного канала ввода-вывода разных потоков данных с разными буферами чревато ошибками при передаче данных. У классов `basic_istream` и `basic_ostream` имеются дополнительные конструкторы для инициализации потока данных с переданным в качестве аргумента буфером. Пример:

```
// io/rdbuf1.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);

    // Переключение между десятичным и шестнадцатеричным выводом
    hexout << "hexout: " << 177 << " ";
    cout    << "cout: "   << 177 << " ";
    hexout << "hexout: " << -49 << " ";
    cout    << "cout: "   << -49 << " ";
    hexout << endl;
}
```

Учтите, что деструктор классов `basic_istream` и `basic_ostream` не удаляет соответствующий потоковый буфер (который не открывается этими классами). Следовательно, чтобы передать поток данных при вызове, вместо ссылки на поток можно передать указатель на потоковый буфер:

```
// io/rdbuf2.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 формат относится к объекту потока данных, а не к буферу. Результат выполнения программы выглядит так:

```
We print 5 lines hexadecimal
0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa
0x2 0x4 0x6 0x8 0xa 0xc 0xe 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, наряду с достоинствами имеется и недостаток — конструирование и уничтожение объекта потока данных обходится дороже, чем простая установка и восстановление форматных флагов. Также следует помнить, что уничтожение объекта потока данных не приводит к очистке буфера. Очищать выходной буфер необходимо «вручную».

Замечание о том, что потоковый буфер не уничтожается, относится только к классам `basic_istream` и `basic_ostream`. Другие потоковые классы уничтожают буфера, созданные ими же, но оставляют буфера, назначенные функцией `rdbuf()` (см. далее).

Перенаправление стандартных потоков данных

В старых реализациях библиотеки `IOStream` глобальные потоки данных `cin`, `cout`, `cerr` и `clog` были объектами классов `istream_withassign` и `ostream_withassign`. Это позволяло перенаправлять потоки данных, присваивая одни потоки другим. Этот механизм был исключен из стандартной библиотеки C++. Тем не менее сама возможность перенаправления потоков данных была сохранена и расширена так, что теперь она может применяться ко всем потокам данных. Перенаправление потока данных осуществляется назначением потокового буфера.

В механизме назначения потоковых буферов перенаправление потоков данных находится под управлением программы, операционная система здесь не участвует. Например, в результате выполнения следующего фрагмента данные, отправленные в поток данных `cout`, будут передаваться не в стандартный выходной канал, а в файл `cout.txt`:

```
std::ofstream file ("cout.txt");
std::cout.rdbuf (file.rdbuf());
```

Для передачи всей форматной информации между потоками данных можно воспользоваться функцией `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, что также приводит к уничтожению соответствующего потокового буфера. В этом отношении файловые потоки данных отличаются от «обычных», поскольку они создают свои объекты потоковых буферов во время конструирования и уничтожают их при уничтожении.

Учитывая это замечание, в приведенном примере дальнейшее использование объекта `cout` для записи невозможно. Более того, его даже нельзя безопасно уничтожить при завершении программы. По этой причине прежний буфер следует всегда сохранять с последующим восстановлением! В представленном ниже примере это делается в функции `redirect()`:

```
// io/redirect.cpp
#include <iostream>
#include <fstream>
using namespace std;

void redirect(ostream&);

int main()
{
    cout << "the first row" << endl;
    redirect(cout);
    cout << "the last row" << endl;
}

void redirect (ostream& strm)
{
    ofstream file("redirect.txt");
    // Сохранение выходного буфера потока
    streambuf* strm_buffer = strm.rdbuf();
```

```
// Перенаправление вывода в файл
strm.rdbuf (file.rdbuf ());

file << "one row for the file" << endl;
strm << "one row for the stream" << endl;

// Восстановление старого выходного буфера
strm.rdbuf (strm_buffer);

} // Автоматическое закрытие файла И буфера
```

Результат выполнения программы выглядит так:

```
the first row
the last row
```

Содержимое файла redirect.txt:

```
one row for the file
one row for the stream
```

Как видите, данные, записанные в поток данных cout внутри функции `redirect()`, были переданы в файл (по имени параметра `strm`), а данные, записанные в `main()` после выполнения `redirect()`, попали в восстановленный выходной канал.

Потоки чтения и записи

В последнем примере связывания один поток данных используется как для чтения, так и для записи. Обычно файл открывается для чтения/записи при помощи класса `fstream`:

```
std::fstream file ("example.txt", std::ios::in | std::ios::out);
```

Также можно использовать два разных потоковых объекта, по одному для чтения и записи. Соответствующий фрагмент может выглядеть примерно так:

```
std::ofstream out ("example.txt", ios::in | ios::out);
std::istream in (out.rdbuf());
```

Объявление `out` открывает файл. Объявление `in` использует потоковый буфер `out` для чтения из него. Обратите внимание: поток данных `out` должен открываться для чтения и записи. Если открыть его только для записи, чтение из потока данных приведет к непредсказуемым последствиям. Также обратите внимание на то, что `in` определяется не с типом `ifstream`, а только с типом `istream`. Файл уже открыт, и у него имеется соответствующий потоковый буфер. Все, что требуется, — это второй потоковый объект. Как и в предыдущих примерах, файл закрывается при уничтожении объекта файлового потока данных.

Также можно создать буфер файлового потока данных и назначить его обоим потоковым объектам. Решение выглядит так:

```
std::filebuf buffer;
std::ostream out(&buffer);
```

```
std::istream in (&buffer);
buffer.open("example.txt", std::ios::in | std::ios::out);
```

Объект `filebuf` является обычной специализацией класса `basic_filebuf<>` для типа `char`. Класс определяет потоковый буфер, используемый файловыми потоками данных.

В следующем примере с помощью цикла в файл выводится четыре строки. После каждой операции вывода все содержимое файла записывается в стандартный выходной поток данных:

```
// iol/rw1.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    // Открытие файла "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++) {
        // Запись одной строки
        output << i << ". line" << endl;

        // Вывод всего содержимого файла
        input.seekg(0);           // Позиционирование в начало
        char c;
        while (input.get(c)) {
            cout.put(c);
        }
        cout << endl;
        input.clear();           // Сброс флагов eofbit и failbit
    }
}
```

Результат выполнения программы выглядит так:

```
1. line
1. line
2. line
1. line
2. line
3. line
1. line
2. line
3. line
4. line
```

Хотя для чтения и записи используются два разных объекта потоков данных, позиции чтения и записи тесно связаны между собой. Функции `seekg()` и `seekp()` вызывают одну и ту же функцию потокового буфера¹. Следовательно, для того чтобы вывести все содержимое файла, необходимо всегда устанавливать позицию чтения в начало файла. После вывода всего содержимого файла позиция чтения/записи снова перемещается в конец файла для присоединения новых строк.

Операции чтения и записи с одним файлом должны разделяться операцией позиционирования (кроме выхода за конец файла во время чтения). Пропуск операции позиционирования приведет к искажению содержимого файла или еще более фатальным ошибкам.

Как упоминалось выше, вместо последовательной обработки символов все содержимое файла можно вывести одной командой, для чего оператору `<<` передается указатель на потоковый буфер:

```
std::cout << input.rebuf();
```

Потоковые классы для работы со строками

Механизм потоковых классов также может использоваться для чтения или записи в строки. У строковых потоков данных имеется буфер, но нет канала ввода-вывода. Для работы с буфером/строкой используются специальные функции. Основная область применения строковых потоков данных — обработка вводимых/выводимых данных независимо от фактического механизма ввода-вывода. Например, выводимый текст можно отформатировать в строке и передать в выходной канал позднее. Другой вариант — ввод данных по строкам и обработка строк с использованием строковых потоков данных.

Исходные потоковые классы для строк в стандартной библиотеке C++ были заменены набором новых классов. Раньше в классах строковых потоков данных для представления строк использовался тип `char*`. Теперь для этой цели используется тип `string` (или в общем случае — `basic_string<>`). Прежние классы строковых потоков данных также являлись частью стандартной библиотеки C++, но сейчас они считаются устаревшими. Они продолжают поддерживаться для обеспечения обратной совместимости, но могут быть исключены из будущих версий стандарта. Прежние классы в новые программы включаться не будут, а в унаследованном коде произойдет их постепенная замена. Тем не менее на с. 623 приводятся краткие описания таких классов.

Классы строковых потоков данных

Для строк определены следующие потоковые классы, которые являются аналогами соответствующих классов файловых потоков данных:

- шаблон `basic_istringstream` со специализациями `istringstream` и `wistringstream` для чтения из строк («строковый входной поток данных»);

¹ В общем случае эта функция различает изменяемую позицию (чтение, запись, обе позиции). Только буфера стандартных потоков данных поддерживают одну позицию и для чтения, и для записи.

- шаблон `basic_ostringstream` со специализациями `ostringstream` и `wostream` для записи в строки («строковый выходной поток данных»);
- шаблон `basic_istringstream` со специализациями `istringstream` и `wistringstream` для чтения и записи в строки;
- шаблон `basic_stringbuf<>` со специализациями `stringbuf` и `wstringbuf` используется другими классами строковых потоков данных для реальных чтения и записи символов.

По аналогии с классами файловых потоков данных эти классы объединены в иерархию, изображенную на рис. 13.3.

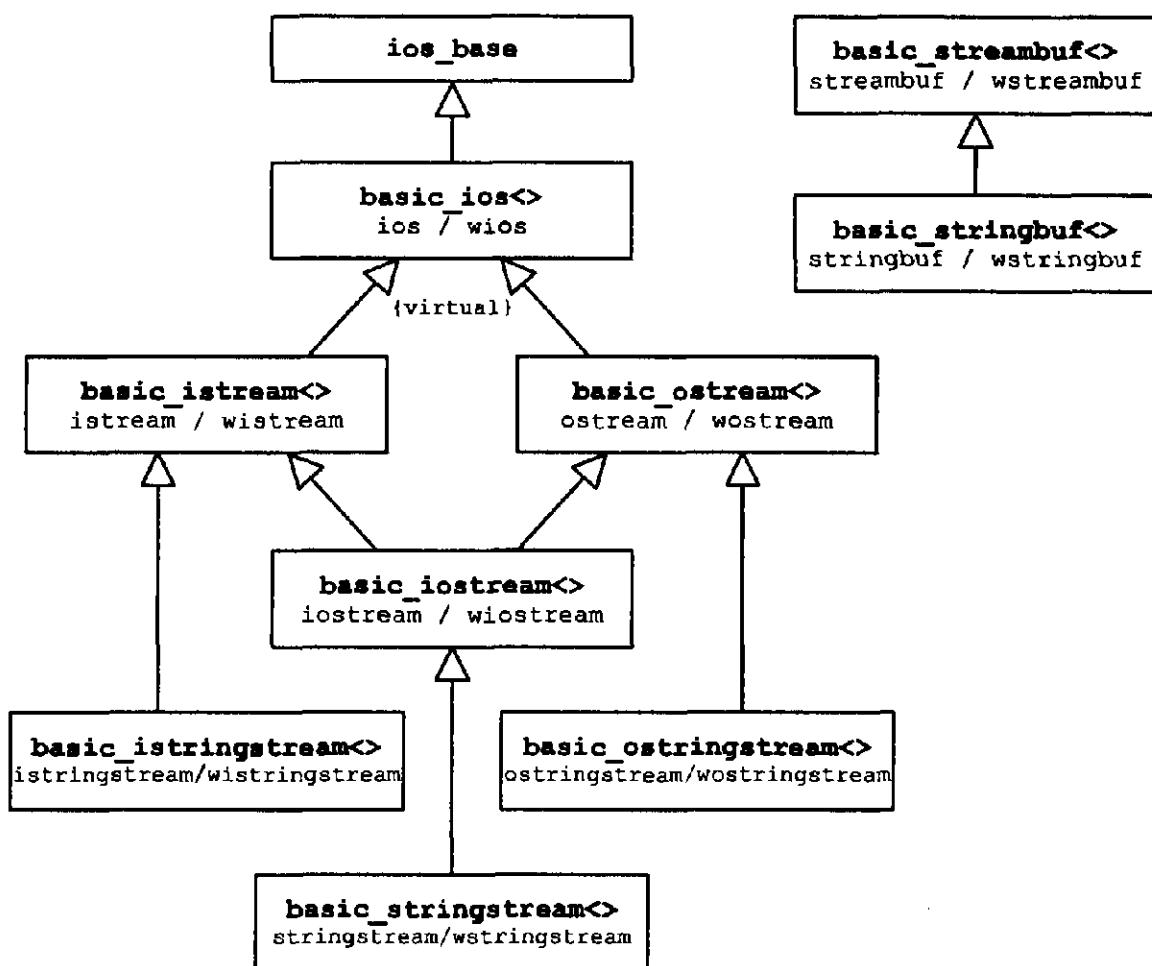


Рис. 13.3. Иерархия классов строковых потоков данных

Эти классы определяются в заголовочном файле `<sstream>` следующим образом:

```

namespace std {
    template <class charT,
              class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
    class basic_istringstream;
    typedef basic_istringstream<char> istringstream;
    typedef basic_istringstream<wchar_t> wistringstream;
}

```

```

template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
    class basic_ostringstream;
typedef basic_ostringstream<char>      ostringstream;
typedef basic_ostringstream<wchar_t>  wostream;

template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
    class basic_stringstream;
typedef basic_stringstream<char>      stringstream;
typedef basic_stringstream<wchar_t>  wstringstream;

template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
    class basic_filebuf;
typedef basic_filebuf<char>      stringbuf;
typedef basic_filebuf<wchar_t>  wstringbuf;
}

```

Главная функция в интерфейсе строковых потоков данных, `str()`, используется для работы с потоковым буфером (табл. 13.39).

Таблица 13.39. Основные операции со строковыми потоками данных

Функция	Описание
<code>str()</code>	Возвращает буфер в виде строки
<code>str(string)</code>	Присваивает <code>string</code> содержимое буфера

Пример использования строковых потоков данных:

```

// io/sstr1.cpp
#include <iostream>
#include <sstream>
#include <bitset>
using namespace std;

int main()
{
    ostringstream os;

    // Десятичное и шестнадцатеричное значение
    os << "dec: " << 15 << hex << " hex: " << 15 << endl;
    cout << os.str() << endl;

    // Присоединение вещественного числа и битового поля
    bitset<15> b(5789);
    os << "float: " << 4.67 << " bitset: " << b << endl;
}

```

```
// Перезапись восьмеричным числом
os.seekp(0);
os << "oct: " << oct << 15;
cout << os.str() << endl;
}
```

Результат выполнения программы выглядит так:

```
dec: 15 hex: f
oct: 17 hex: f
float: 4.67 bitset: 001011010011101
```

Сначала в `os` выводятся два числа, десятичное и шестнадцатеричное. Затем к ним присоединяется вещественное число и битовое поле (в двоичном виде). Функция `seekp()` перемещает позицию записи в начало потока данных, поэтому при следующем вызове оператора `<<` данные записываются в начало строки с перезаписью существующего строкового потока данных. Тем не менее символы, которые не были перезаписаны, остаются действительными. Если вы хотите полностью удалить текущее содержимое потока данных, присвойте новое содержимое буфера функцией `str()`:

```
strm.str("");
```

Первые строки, записанные в `os`, завершаются манипулятором `endl`. Это означает, что в конце вывода производится перевод строки. Поскольку за строкой в поток данных также выводится манипулятор `endl`, в результате будут записаны два символа новой строки подряд. Этим объясняется наличие пустых строк в выходных данных.

При работе со строковыми потоками данных программисты часто допускают распространенную ошибку — они забывает извлечь строку функцией `str()` и вместо этого выводят объект потока данных. С точки зрения компилятора это вполне разумно и возможно, поскольку существует преобразование к `void*`. В результате состояние потока данных выводится в виде адреса (см. с. 571).

Одним из типичных применений строковых выходных потоков данных является определение операторов вывода для пользовательских типов (см. с. 626).

Строчные входные потоки данных в основном используются для формированного чтения из существующих строк. Например, часто бывает проще читать данные по строкам и затем анализировать каждую строку в отдельности. В следующем фрагменте из строки `s` читается целое число `x`, равное 3, и вещественное число `f`, равное 0.7:

```
int x;
float f;
std::string s = "3.7";

std::istringstream is(s);
is >> x >> f;
```

При создании строковых потоков данных могут задаваться флаги режима открытия файла (см. с. 606) и/или существующие строки. С флагами `ios::app`

и `ios::ate` символы, записанные в строковый поток данных, присоединяются к существующей строке:

```
std::string s;  
...  
std::ostringstream os (s, ios::out|ios||app);  
os << 77 << std::hex << 77;
```

Однако строка, возвращаемая функцией `str()`, представляет собой копию строки `s` с присоединенными значениями `77` в десятичном и шестнадцатеричном виде. Сама строка `s` остается неизменной.

Потоковые классы `char*`

Потоковые классы `char*` поддерживаются только в целях обратной совместимости. Их интерфейс иногда порождает ошибки, с этими классами часто работают неправильно. Тем не менее они продолжают широко использоваться, поэтому далее приводятся их краткие описания. Учтите, что в описываемой здесь стандартной версии прежний интерфейс был слегка изменен.

Далее вместо термина *строка* будет использоваться термин *последовательность символов*. Дело в том, что последовательность символов, поддерживаемая потоковыми классами `char*`, не всегда заканчивается символом завершения строки (а следовательно, не является строкой в обычном понимании).

Потоковые классы `char*` определены только для символьного типа `char`. К этой категории относятся следующие классы:

- `istrstream` для чтения из последовательностей символов;
- `ostrstream` для записи в последовательности символов;
- `strstream` для чтения из последовательностей символов и записи в последовательности символов;
- `strstreambuf` используется как потоковый буфер для потоков `char*`.

Потоковые классы `char*` определяются в заголовочном файле `<strstream>`.

Поток данных `istrstream` может инициализироваться последовательностью символов (типа `char*`), либо завершенной символом `\0`, либо имеющей длину, переданную в аргументе. Типичный пример чтения и обработки строк:

```
char buffer[1000];      // Буфер, в котором могут храниться  
                      // до 999 символов  
// Чтение строки  
std::cin.get(buffer,sizeof(buffer));  
  
// Чтение/обработка строки как потока  
std::istrstream input(buffer);  
  
...  
input >> x;
```

Поток данных `char*`, предназначенный для записи, поддерживает последовательность символов, растущую по мере необходимости, или инициализируется

буфером фиксированного размера. При помощи флагов `ios::app` и `ios::ate` можно дописывать выводимые символы к последовательности, уже хранящейся в буфере.

При строковой интерпретации потока данных `char*` необходима осторожность. В отличие от строковых потоков данных потоки типа `char*` не всегда следят за использованием памяти, в которой хранится последовательность символов.

Функция `str()` предоставляет вызывающей стороне доступ к последовательности символов вместе с ответственностью за управление соответствующей памятью. Если поток данных не был инициализирован буфером фиксированного размера (за который поток никогда не отвечает), должны соблюдаться следующие три правила.

- Поскольку права владения памятью передаются вызывающей стороне (если поток данных не был инициализирован буфером фиксированного размера), символьная последовательность должна освобождаться. Но так как не гарантируется, что память выделялась¹, вызов `delete[]` не всегда безопасен. Лучше всего вернуть память в поток данных вызовом функции `freeze` с аргументом `false` (пример приводится далее).
- Вызов `str()` запрещает потоку данных дальнейшее изменение последовательности символов. Функция неявно вызывает функцию `freeze()`, которая фиксирует («замораживает») последовательность символов. Это делается для того, чтобы избежать трудностей при недостаточном размере буфера и необходимости выделять новую память.
- Функция `str()` не присоединяет символ завершения строки ('\0'). Это символ приходится отдельно присоединять к потоку данных для завершения последовательности символов. Задача решается при помощи манипулятора `ends`. Некоторые реализации присоединяют символ завершения строки автоматически, но это поведение нарушает переносимость программы.

Пример использования потока `char*`:

```
float x;
...
/* Создание и заполнение потока char*
 * - не забывайте о ends или '\0'!!!
 */
std::ostrstream buffer;      // Динамический потоковый буфер
buffer << "float x: " << x << std::ends;

// Передача полученной С-строки функции foo() и возвращение памяти в буфер
char* s = buffer.str();
foo(s);
buffer.freeze(false);
```

Зафиксированный поток `char*` можно привести в обычное состояние для дополнительной обработки. Для этого следует вызвать функцию `freeze()` с аргу-

¹ Имеется конструктор, в аргументах которого передаются два указателя на функции: для выделения и освобождения памяти.

ментом `false`. При выполнении этой операции право владения последовательностью символов снова возвращается объекту потока данных. Это единственный безопасный способ освобождения памяти для последовательности символов. В следующем примере показано, как это делается:

```
float x;  
...  
std::ostrstream buffer; // Динамический поток char*  
  
// Заполнение потока char*  
buffer << "float x: " << x << std::endl;  
  
/* Передача итоговой С-строки функции foo()  
 * - фиксация потока char*  
 */  
foo(buffer.str());  
  
// Снятие фиксации с потока char*  
buffer.freeze(false);  
  
// Установка позиции записи в начало  
buffer.seekp(0, ios::beg);  
  
// Повторное заполнение потока char*  
buffer << "once more float x: " << x << std::endl;  
  
/* Повторная передача полученной С-строки функции foo()  
 * - фиксация потока char*;  
 */  
foo(buffer.str());  
  
// Возвращение памяти в буфер  
buffer.freeze(false);
```

Проблем, возникающих из-за фиксации потока данных, нет в классах строковых потоков данных. В основном это объясняется тем, что использование памяти при копировании находится под контролем строкового класса.

Операторы ввода-вывода для пользовательских типов

Как упоминалось ранее, главным преимуществом потокового ввода-вывода перед средствами ввода-вывода языка С является возможность расширения потокового механизма для пользовательских типов. Расширение основано на перегрузке операторов `<<` и `>>`. Далее рассматривается пример использования потоков данных для вывода правильных дробей.

Реализация операторов вывода

В выражении с оператором вывода `<<` левый операнд определяет поток данных, а правый — объект, записываемый в этот поток:

`поток << объект`

В соответствии с правилами языка эта конструкция может интерпретироваться двумя способами:

`поток.operator<<(объект)`
`поток.operator<<(поток, объект)`

Первая интерпретация используется для встроенных типов. Для пользовательских типов должна использоваться вторая интерпретация, поскольку потоковые классы закрыты для расширения. Все, что требуется, — реализовать глобальный оператор `<<` для пользовательских типов. Задача решается относительно просто, если при этом не нужен доступ к закрытым членам объекта (но об этом позже).

Например, для вывода объекта класса `Fraction` в формате *числитель/знаменатель* можно воспользоваться следующей функцией:

```
// io/fracout.cpp
#include <iostream>

inline
std::ostream& operator << (std::ostream& strm, const Fraction& f)
{
    strm << f.numerator() << '/' << f.denominator();
    return strm;
}
```

Функция выводит числитель и знаменатель, разделенные символом `/`, в поток данных, передаваемый в аргументе, — файловый, строковый или еще какой-либо. Для поддержки цепочечных операций вывода, а также для совмещения вывода с проверкой состояния потока данных функция возвращает ссылку на поток.

У этой простой формы есть два основных недостатка.

- Из-за использования в сигнатуре класса `ostream` функция применима только к потокам данных с типом символов `char`. Если функция предназначается только для Европы и Америки, проблем не будет. С другой стороны, построить более универсальную версию совсем несложно, поэтому следует по крайней мере рассмотреть такую возможность.
- 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, то есть в данном случае — к выводу числителя. Пример:

```
Fraction vat(16,100); // В Германии действует единая ставка НДС=16%
std::cout << "VAT: \" " << std::left << std::setw(8)
    << vat << ' "' << std::endl;
```

Эта программа выведет следующий результат:

VAT: "16 /100"

В следующей версии решены обе проблемы:

```
// io/frac2out.cpp
#include <iostream>
#include <sstream>

template <class charT, class 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);

    // Заполнение строкового потока
    s << f.numerator() << '/' << f.denominator();

    // print string stream
    strm << s.str();

    return strm;
}
```

Оператор превратился в шаблон функции, параметризованный для всех разновидностей потоков данных. Проблема с шириной поля решается записью в строковый поток данных без указания конкретной ширины. Сконструированная строка затем передается в поток данных, переданный в аргументе. В результате символьное представление дроби выводится одной операцией записи, к которой применяется ширина поля. Например, рассмотрим такой фрагмент:

```
Fraction vat(16,100);    // В Германии действует единая ставка НДС=16%
std::cout << "VAT: \""
        << std::left << std::setw(8)
        << vat << '\"' << std::endl;
```

Этот фрагмент выведет следующий результат:

VAT: "16/100 "

Реализация операторов ввода

Операторы ввода реализуются по тому же принципу, что и операторы вывода. Тем не менее при вводе приходится учитывать возможные ошибки чтения. Обычно

в функциях ввода предусматривается особая обработка ситуаций, когда ввод завершается неудачей.

При реализации функции ввода приходится выбирать между простотой и гибкостью. Например, в следующей функции используется упрощенный подход — дробь читается без проверки возможных ошибок:

```
// io.fraclin.cpp
#include <iostream>

inline
std::istream& operator >> (std::istream& strm, Fraction& f)
{
    int n, d;

    strm >> n;      // Ввод числителя
    strm.ignore();   // Пропуск '/'
    strm >> d;      // Ввод знаменателя

    f = Fraction(n,d); // Присваивание всей дроби

    return strm;
}
```

Во-первых, такая реализация подходит только для потоков данных с типом символов `char`. Во-вторых, она не проверяет, действительно ли два числа разделяются символом `/`.

Другая проблема возникает при вводе неопределенных значений. Если знаменатель прочитанной дроби равен 0, ее поведение не определено. Проблема обнаруживается в конструкторе класса `Fraction`, вызываемом выражением `Fraction(n,d)`. Но это означает, что ошибки форматирования автоматически обрабатываются внутри класса `Fraction`. Так как на практике ошибки форматирования обычно регистрируются на уровне потоков данных, лучше установить в этом случае флаг `ios_base::failbit`.

Наконец, даже неудачная операция чтения может модифицировать дробь, переданную по ссылке. Допустим, числитель был прочитан успешно, а при чтении знаменателя произошла ошибка. Такое поведение противоречит общепринятым правилам, установленным стандартными операторами ввода, и поэтому считается нежелательным. Операция чтения должна либо завершаться успешно, либо не вносить изменений.

Ниже приведена усовершенствованная реализация программы, избавленная от этих недостатков. Кроме того, она более универсальна, поскольку благодаря параметризации подходит для любых типов потоков данных:

```
// io/frac2in.hpp
#include <iostream>

template <class charT, class traits>
inline
std::basic_istream<charT,traits>&
```

```
operator >> (std::basic_istream<charT,traits>& strm, Fraction& f)
{
    int n, d;

    // Ввод числителя
    strm >> n;

    /* Если числитель прочитан успешно
     * - прочитать '/' и знаменатель
     */
    if (strm.peek() == '/') {
        strm.ignore();
        strm >> d;
    }
    else {
        d = 1;
    }

    /* Если знаменатель равен нулю
     * - установить failbit как признак ошибки форматирования ввода-вывода
     */
    if (d == 0) {
        strm.setstate(std::ios::failbit);
        return strm;
    }

    /* Если все прошло успешно,
     * изменить значение дроби
     */
    if (strm) {
        f = Fraction(n,d);
    }

    return strm;
}
```

На этот раз знаменатель читается только в том случае, если за первым числом следует символ /; в противном случае по умолчанию используется знаменатель, равный 1, а целое число интерпретируется как дробь. Таким образом, для целых чисел знаменатель не обязателен.

Реализация также проверяет, не равен ли прочитанный знаменатель нулю. В этом случае устанавливается флаг `ios_base::failbit`, что может привести к выдаче соответствующего исключения (см. с. 576). Разумеется, при нулевом знаменателе возможны и другие действия. Например, реализация может сама сгенерировать исключение или вообще отказаться от проверки знаменателя, чтобы исключение было сгенерировано классом `Fraction`.

Наконец, мы проверяем состояние потока данных, и новое значение присваивается объекту дроби только в том случае, если ввод был выполнен без ошибок. Всегда выполняйте последнюю проверку и изменяйте значение объекта, если чтение прошло успешно.

Конечно, разумность чтения целых чисел как дробей можно поставить под сомнение. Существуют и другие нюансы, которые тоже можно исправить: например, символ \ должен следовать за числителем без разделяющих пробелов. С другой стороны, знаменателю может предшествовать произвольное количество пробелов, которые обычно игнорируются. Впрочем, это дает лишь отдаленное представление о сложностях, связанных с чтением нетривиальных структур данных.

Ввод-вывод с использованием вспомогательных функций

Если реализация оператора ввода-вывода требует доступа к закрытым данным объекта, то стандартные операторы должны поручить фактическую работу вспомогательным функциям классов. При помощи такого подхода можно также реализовать полиморфные чтение и запись. Примерная реализация выглядит так:

```
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;
}
```

Типичный пример — непосредственный доступ к числителю и знаменателю дроби при вводе:

```
void Fraction::scanFrom (std::istream& strm)
{
    ...
    // Прямое присваивание значений компонентов
    num = n;
    denom = d;
}
```

Существует и другое решение: если класс не будет использоваться в качестве базового, операторы ввода-вывода можно объявить дружественными (*friend*) для

данного класса. Однако этот подход существенно ограничивает свободу действий при наследовании. Дружественные функции не могут быть виртуальными, что является потенциальной причиной ошибочных вызовов. Например, если в аргументе оператора ввода передается ссылка на базовый класс, которая на самом деле ссылается на объект производного класса, то для нее будет вызван оператор базового класса. Для решения проблемы производные классы не должны реализовывать собственные операторы ввода-вывода. Таким образом, представленная реализация более универсальна, чем дружественные функции. Именно она может рассматриваться как стандартное решение, хотя в большинстве примеров применяются дружественные функции.

Пользовательские операторы с функциями неформатированного ввода-вывода

Приведенные выше реализации операторов ввода-вывода поручали основную работу готовым операторам форматированного ввода-вывода. Иначе говоря, операторы << и >> реализовывались в контексте соответствующих операторов более простых типов.

Операторы ввода-вывода в стандартной библиотеке C++ определяются иначе. Общая схема выглядит так: сначала поток данных проходит предварительную обработку и готовится к вводу-выводу. Затем происходит собственно ввод-вывод и некоторая завершающая обработка. Используйте эту схему при определении собственных операторов ввода-вывода, тем самым будет обеспечена их логическая согласованность.

В классах `basic_istream` и `basic_ostream` определяется вспомогательный класс `sentry`. Конструктор этого класса выполняет предварительную обработку, а деструктор — соответствующие завершающие действия. Эти классы заменяют функции, использовавшиеся в предыдущих реализациях библиотеки `IOStream` (`ipfx()`, `isfx()`, `opfx()` и `osfx()`). Новая схема обеспечивает завершающую обработку даже в том случае, если ввод-вывод будет отменен из-за исключения.

Если оператор ввода-вывода использует функцию неформатированного ввода-вывода или напрямую работает с потоковым буфером, для него прежде всего необходимо сконструировать объект `sentry`. Последующая обработка будет зависеть от состояния этого объекта, по которому проверяется состояние потока данных. Для этой цели объект `sentry` обычно преобразуется к типу `bool`. Следовательно, операторы ввода-вывода в общем виде выглядят так:

```
sentry se(strm): // Косвенная организация
                  // предварительной и завершающей обработки
if (se) {
    ...
    // Собственно работа с потоком
}
```

В аргументе конструктора класс `sentry` получает объект `strm`, для которого должны выполняться предварительный и завершающий этапы обработки.

Также необходимо дополнительно позаботиться о решении общих задач операторов ввода-вывода (синхронизации потоков данных, проверки нормального

состояния потока данных, игнорировании пропусков и т. д.), а также дополнительных задач, зависящих от реализации. Например, в многопоточных средах, то есть в средах с несколькими параллельно функционирующими потоками выполнения (*threads*), такой задачей может быть блокировка доступа.

Для входных потоков данных при конструировании объекта **sentry** может передаваться необязательный логический признак, который указывает, что пропуски должны читаться даже при установленном флаге **skipws**:

```
sentry se(strm,true); // Не игнорировать пропуски при чтении
```

Следующий пример демонстрирует эту возможность для класса **Row**, представляющего строку в текстовом редакторе.

- Оператор << выводит строку функцией **write()** класса потока данных:

```
std::ostream& operator<< (std::ostream& strm, const Row& row)
{
    // Организация предварительной и завершающей обработки
    std::ostream::sentry se(strm);
    if (se) {
        // Выполнение вывода
        strm.write(row.c_str(),row.len());
    }

    return strm;
}
```

- Оператор >> в цикле читает строку по символам. Конструктору объекта **sentry** передается аргумент **true**, чтобы пропуски не игнорировались при вводе:

```
std::istream& operator>> (std::istream& strm, Row& row)
{
    /* Организация предварительной и завершающей обработки
     * - true: пропуски не игнорируются
     */
    std::istream::sentry se(strm,true);
    if (se) {
        // Выполнение ввода
        char c;
        row.clear();
        while (strm.get(c) && c != '\n') {
            row.append(c);
        }
    }

    return strm;
}
```

Очевидно, что эта архитектура может использоваться даже в том случае, если реализация основана не на вызове функций неформатированного ввода-вывода, а на применении операторов. Тем не менее использование членов классов **basic_istream** или **basic_ostream** для чтения или записи символов внутри кода, за-

щищенного объектами `sentry`, малоэффективно. По возможности следует использовать соответствующий объект `basic_streambuf`.

Пользовательские форматные флаги

При определении пользовательских операторов ввода-вывода часто бывает удобно определить для этих операторов специальные форматные флаги (вероятно, устанавливаемые при помощи соответствующих манипуляторов). Например, было бы неплохо, если бы приведенный выше оператор вывода дробей можно было настроить на отделение пробелами числителя и знаменателя от символа `/`.

Объекты потоков данных предоставляют такую возможность — в них предусмотрен механизм связывания данных с потоком. Он позволяет задать нужные значения (например, при помощи 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&` и `void*&`. Предполагается, что `iword()` и `pword()` обращаются к объектам `long` или `void*` в массиве произвольного размера, хранящемся в объекте потока данных. Форматные флаги, сохраняемые для потока, располагаются по одному и тому же индексу для всех потоков. Статическая функция `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 интерпретироваться и как представление форматирования по умолчанию, и как признак того, что данные еще не инициализировались. Пример:

```
// Получение индекса для новых данных ostream
static const int iword_index = std::ios_base::xalloc();

// Определение манипулятора для модификации 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)
{
    /* Запросить данные у ostream
     * - true: использовать пробелы между числителем и знаменателем
     * - false: выводить без пробелов
     */
    if (strm.iword(iword_index)) {
        strm << f.numerator() << " / " << f.denominator();
    }
    else {
        strm << f.numerator() << "/" << f.denominator();
    }
    return strm;
}
```

В этом примере используется упрощенный подход к реализации оператора вывода, поскольку его основная цель — демонстрация функции `lword()`. Форматный флаг считается логическим признаком, определяющим необходимость вывода пробелов между числителем и знаменателем.

В первой строке функция `ios_base::xalloc()` возвращает индекс, который может использоваться для хранения форматного флага. Результат вызова сохраняется в константе, так как это значение не модифицируется. Функция `fraction_spaces()` представляет собой манипулятор для установки значения `Int`, хранящегося по индексу `lword_Index` в целочисленном массиве, связанном с потоком данных `strm`. Оператор вывода получает это значение и выводит дробь в соответствии с состоянием флага. Если флаг равен `false`, по умолчанию дробь выводится без пробелов, а если нет — символ / окружается двумя пробелами.

Функции `lword()` и `pword()` возвращают ссылки на `Int` или `void*`. Эти ссылки остаются действительными только до следующего вызова `iword()` или `pword()` с соответствующим объектом потока данных или до уничтожения объекта потока. Обычно результаты `iword()` и `pword()` сохраняться не должны. Предполагается, что доступ происходит достаточно быстро, хотя представление данных в виде массива не гарантировано.

Функция `copyfmt()` копирует всю форматирующую информацию (см. с. 591), в том числе и массивы, с которыми работают функции `iword()` и `pword()`. Это может вызвать проблемы для объектов, сохраняемых в контексте потока данных при помощи `pword()`. Например, если значение представляет собой адрес объекта, то вместо объекта будет скопирован только адрес. Как следствие, смена формата в одном потоке данных будет распространяться на другие потоки. Также может быть желательно, чтобы объект, ассоциированный с потоком данных функцией `pword()`, уничтожался при уничтожении потока. Следовательно, для таких объектов желательно реализовать «глубокое» копирование вместо «поверхностного».

Для подобных целей (например, реализации глубокого копирования в случае необходимости или удаления объекта при уничтожении потока данных) в `ios_base` определен механизм обратного вызова. Функция `register_callback()` регистрирует функцию, вызываемую при выполнении определенных условий для объекта `ios_base`. Функция определяется следующим образом:

```
namespace std {
    class ios_base {
        public:
            // Разновидности событий обратного вызова
            enum event { erase_event, imbue_event, copyfmt_event };
            // Тип функции обратного вызова
            typedef void (*event_callback)(event e, ios_base& strm,
                                         int arg);
            // Регистрация функций обратного вызова
            void register_callback(event_callback cb, int arg);
            ...
    };
}
```

Функция `register_callback()` получает два аргумента: указатель на функцию и число `Int`. Аргумент `Int` передается в третьем аргументе при вызове зарегистрированной функции. Например, он может использоваться для идентификации индекса `pword()`, то есть определять обрабатываемый элемент массива. Аргумент `strm`, передаваемый функции обратного вызова, содержит объект `ios_base`, обратившийся к этой функции. Аргумент `e` определяет причину вызова, его допустимые значения перечислены в табл. 13.40.

Таблица 13.40. Причины обратного вызова

Событие	Причина
<code>ios_base::imbue_event</code>	Задание локального контекста функцией <code>imbue()</code>
<code>ios_base::erase_event</code>	Уничтожение потока или использование <code>copyfmt()</code>
<code>ios_base::copy_event</code>	Использование функции <code>copyfmt()</code>

При вызове для объекта функции `copyfmt()` функции обратного вызова вызываются дважды. Еще до начала копирования они вызываются с аргументом `erase_event` для выполнения необходимой зачистки (например, удаления объектов, хранящихся в массиве `pword()`). После копирования данных форматирования функции обратного вызова вызываются снова, на этот раз — с аргументом `copy_event`. Например, этот проход может использоваться для организации глубокого копирования объектов, хранящихся в массиве `pword()`. Обратите внимание: вместе с данными форматирования копируются и функции обратного вызова, а исходный список зарегистрированных функций удаляется. Следовательно, при втором проходе будут вызваны только что скопированные функции.

Механизм обратного вызова чрезвычайно примитивен. Он не позволяет отменять регистрацию функций обратного вызова (не считая вызова `copyfmt()` с аргументом, не имеющим зарегистрированных функций). Кроме того, повторная регистрация функции обратного вызова даже с тем же аргументом приведет к повторному вызову. Тем не менее библиотека гарантирует, что функции будут вызваны в порядке, обратном порядку их регистрации. Это сделано для того, чтобы функция обратного вызова, зарегистрированная из другой функции, не вызывалась до следующего срабатывания функций обратного вызова.

Правила построения пользовательских операторов ввода-вывода

Ниже перечислены некоторые правила, которые должны соблюдаться в пользовательских реализациях операторов ввода-вывода. Эти правила продиктованы типичным поведением стандартных операторов.

- Выходной формат должен допускать определение оператора ввода, читающего данные без потери информации. В некоторых случаях — особенно 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 форматная спецификация потока данных. Прежде всего это относится к ширине поля при выводе.
- При возникновении ошибок должен быть установлен соответствующий флаг состояния.
- Ошибки не должны изменять состояние объекта. Если оператор читает несколько объектов данных, промежуточные результаты сохраняются во временных объектах до окончательного принятия значения.
- Вывод не должен завершаться символом новой строки, в основном из-за того, что это не позволит вывести другие объекты в той же строке.
- Даже слишком большие данные должны читаться полностью. После чтения следует установить соответствующий флаг ошибки, а возвращаемое значение должно содержать полезную информацию (например, максимальное допустимое значение).
- При обнаружении ошибки форматирования следует по возможности прекратить чтение.

Классы потоковых буферов

Как упоминалось на с. 564, потоки данных не выполняют непосредственные операции чтения и записи, а поручают их потоковым буферам. Этот раздел посвящен работе классов буферов. Приведенный материал не только поможет глубже разобраться в том, как работают потоки данных при вводе-выводе, но и заложит основу для определения новых каналов ввода-вывода. Прежде чем переходить к подробному описанию потоковых буферов, рассмотрим их открытый интерфейс.

Потоковые буферы с точки зрения пользователя

С точки зрения пользователя потокового буфера, класс `basic_streambuf` представляет собой нечто, принимающее и отправляющее символы. В табл. 13.41 приведены версии открытых функций для вывода символов.

Таблица 13.41. Открытые функции вывода символов

Функция	Описание
<code>sputc(c)</code>	Выводит символ <code>c</code> в потоковый буфер
<code>sputc(s, n)</code>	Выводит <code>n</code> символов из последовательности <code>s</code> в потоковый буфер

Функция `sputc()` возвращает `traits_type::eof()` в случае ошибки, где `traits_type` – определение типа в классе `basic_streambuf`. Функция `sputn()` выводит количество символов, заданное вторым аргументом, если только выводу не помешают недостаточные размеры строкового буфера. Символы завершения строк при выводе не учитываются. Функция возвращает количество выведенных символов.

Интерфейс чтения символов из потокового буфера устроен несколько сложнее (табл. 13.42). Дело в том, что при вводе иногда требуется узнать символ без его извлечения из буфера. Кроме того, желательно предусмотреть возможность возврата символов в потоковый буфер. Соответствующие функции поддерживаются классами потоковых буферов.

Таблица 13.42. Открытые функции ввода символов

Функция	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)	Читает n символов и сохраняет их в буфере b
sputbackc(c)	Возвращает символ c в потоковый буфер
sungetc()	Возвращается на одну позицию 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 значения, возвращаемого этой функцией.

Пока потоковый буфер не достигнет конца потока данных, один из символов считается «текущим». Функция `sgetc()` используется для получения текущего символа без перемещения к следующему символу. Функция `sbumpc()` читает текущий символ и переходит к следующему символу, который становится текущим. Последняя из функций чтения отдельных символов, `snextc()`, переходит к следующему символу и читает новый текущий символ. Для обозначения неудачи все три функции возвращают `traits_type::eof()`. Функция `sgetn()` читает в буфер последовательность символов, максимальная длина которой передается в аргументе. Функция возвращает количество прочитанных символов.

Функции `sputbackc()` и `sungetc()` возвращаются на одну позицию в потоке данных, вследствие чего текущим становится предыдущий символ. Функция `sputbackc()` может использоваться для замены предыдущего символа другим символом. При вызове этих функций необходимо соблюдать осторожность: нередко возврат ограничивается всего одним символом.

Отдельная группа функций используется для подключения объекта локального контекста, для смены позиции и управления буферизацией. Эти функции перечислены в табл. 13.43.

Функции `pubimbue()` и `getloc()` используются при интернационализации (см. с. 601). Функция `pubimbue()` подключает новый объект локального контекста к потоковому буферу и возвращает ранее установленный объект локального контекста. Функция `getloc()` возвращает текущий объект локального контекста.

Таблица 13.43. Прочие функции открытого интерфейса потоковых буферов

Функция	Описание
pubimbue(loc)	Ассоциирует потоковый буфер с локальным контекстом loc
getloc()	Возвращает текущий локальный контекст
pubseekpos(pos)	Перемещает текущую позицию в заданную абсолютную позицию
pubseekpos(pos, which)	То же с указанием направления ввода-вывода
pubseekoff(offset, rpos)	Перемещает текущую позицию по отношению к другой позиции
pubseekoff(offset, rpos, whlch)	То же с указанием направления ввода-вывода
pubsetbuf(b,n)	Управление буферизацией

Функция `pubsetbuf()` позволяет в определенной степени управлять стратегией буферизации потоковых буферов. Тем не менее ее возможности зависят от конкретного класса потокового буфера. Например, вызов функции `pubsetbuf()` для буферов строковых потоков данных не имеет смысла. Даже для буферов файловых потоков данных применение этой функции переносимо лишь в том случае, если она вызывается перед выполнением первой операции ввода-вывода в формате `pubsetbuf(0,0)` (то есть буфер не используется). Функция возвращает объект потокового буфера при успешном завершении или 0 в случае неудачи.

Функции `pubseekoff()` и `pubseekpos()` используются для управления текущей позицией чтения и/или записи. Позиция зависит от последнего аргумента, который относится к типу `ios_base::openmode` и по умолчанию равен `ios_base::in | ios_base::out`. При установленном флаге `ios_base::in` изменяется позиция чтения, а при установленном флаге `ios_base::in` — позиция записи. Функция `pubseekpos()` перемещает поток данных в абсолютную позицию, заданную первым аргументом, тогда как 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` (за подробностями обращайтесь на с. 610). Обе функции возвращают новую текущую позицию или признак недействительной позиции. Чтобы обнаружить недействительную 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. с. 609). Текущая позиция потока возвращается функцией `pubseekoff()`:

```
sbuff.pubseekoff(0, std::ios::cur)
```

Итераторы потоковых буферов

Другой механизм неформатированного ввода-вывода основан на использовании классов итераторов потоковых буферов. Эти классы удовлетворяют требованиям к итераторам ввода и вывода, предназначенным для чтения или записи отдельных символов в потоковых буферах, и совместимы со средствами посимвольного ввода-вывода алгоритмов стандартной библиотеки C++.

Шаблоны `istreambuf_iterator` и `ostreambuf_iterator` используются для чтения или записи отдельных символов с объектами типа `basic_streambuf`. Определения этих классов в заголовочном файле `<iterator>` выглядят примерно так:

```
namespace std {
    template <class charT,
              class traits = char_traits<charT> >
    istreambuf_iterator;
    template <class charT,
              class traits = char_traits<charT> >
    ostreambuf_iterator;
}
```

Эти итераторы представляют собой специализированные версии потоковых итераторов, описанных на с. 281. Единственное отличие заключается в том, что их элементы относятся к символьному типу.

Итераторы потоковых буферов вывода

Вывод строки в потоковый буфер с использованием итератора `ostreambuf_iterator` выполняется так:

```
// Создание итератора для буфера или выходного потока cout
std::ostreambuf_iterator<char> bufWriter(std::cout);

std::string hello("hello, world\n");
std::copy(hello.begin(), hello.end(),
          bufWriter);
```

В первой строке этого фрагмента конструируется итератор вывода типа `ostreambuf_iterator` для объекта `cout`. Вместо передачи выходного потока данных можно сразу передать указатель на потоковый буфер. Остальные команды конструируют объект `string` и копируют символы этого объекта через сконструированный итератор вывода.

В табл. 13.44 перечислены все операции итераторов потоковых буферов вывода. Этот интерфейс имеет много общего с потоковыми итераторами вывода (см. с. 282). Кроме того, итератор можно инициализировать буфером, а также проверить возможность записи через итератор функцией `failed()`. Если предыдущий вывод символов завершился неудачей, `failed()` возвращает `true`. В этом случае любые попытки записи оператором `=` ни к чему не приводят.

Таблица 13.44. Операции итераторов потоковых буферов вывода

Выражение	Описание
<code>ostreambuf_iterator<char>(ostream)</code>	Создание итератора потокового буфера вывода для потока <code>ostream</code>
<code>ostreambuf_iterator<char>(buffer_ptr)</code>	Создание итератора потокового буфера вывода для буфера, на который ссылается указатель <code>buffer_ptr</code>
<code>*iter</code>	Фиктивная операция (возвращает <code>iter</code>)

продолжение ↓

Таблица 13.44 (продолжение)

Выражение	Описание
<code>iter = c</code>	Записывает символ <code>c</code> в буфер вызовом функции <code>sputc(c)</code>
<code>++iter</code>	Фиктивная операция (возвращает <code>iter</code>)
<code>iter++</code>	Фиктивная операция (возвращает <code>iter</code>)
<code>failed()</code>	Проверка возможности дальнейшей записи через итератор потокового буфера вывода

Итераторы потоковых буферов ввода

В табл. 13.45 перечислены все операции итераторов потоковых буферов ввода. Этот интерфейс имеет много общего с потоковыми итераторами ввода (см. с. 284). Кроме того, итератор можно инициализировать буфером, а также проверить равенство двух итераторов потоковых буферов ввода функцией `equal()`. Два итератора потоковых буферов ввода равны, если они оба установлены в конец потока данных или ни один из них не установлен в конец потока данных.

Таблица 13.45. Операции итераторов потоковых буферов ввода

Выражение	Описание
<code>istreambuf_iterator<char>()</code>	Создает итератор конца потока
<code>istreambuf_iterator<char>(istream)</code>	Создает итератор потокового буфера ввода для потока <code>istream</code> , возможно, с чтением первого символа функцией <code>sgetc()</code>
<code>istreambuf_iterator<char>(buffer_ptr)</code>	Создает итератор потокового буфера ввода для буфера, на который ссылается указатель <code>buffer_ptr</code> , возможно, с чтением первого символа функцией <code>sgetc()</code>
<code>*iter</code>	Возвращает текущий символ, ранее прочитанный функцией <code>sgetc()</code> (читает первый символ, если он не был прочитан конструктором)
<code>++iter</code>	Читает следующий символ функцией <code>sbufpc()</code> и возвращает его позицию
<code>iter++</code>	Читает следующий символ функцией <code>sbufpc()</code> , но возвращает итератор для предыдущего символа
<code>iter1.equal(iter2)</code>	Проверяет равенство двух итераторов
<code>iter1==iter2</code>	Проверка на равенство <code>iter1</code> и <code>iter2</code>
<code>iter1!=iter2</code>	Проверка на неравенство <code>iter1</code> и <code>iter2</code>

Из этого следует несколько неочевидная формулировка эквивалентности объектов типа `istreambuf_iterator`. Два объекта типа `istreambuf_iterator` считаются эквивалентными, если оба итератора установлены в конец потока данных или ни один из них не установлен в конец потока данных (совпадают ли при этом буфера — значения не имеет). В частности, итератор, установленный в конец потока данных, может быть получен при конструировании итератора конструк-

тором по умолчанию. Кроме того, итератор `istreambuf_iterator` устанавливается в конец потока данных при попытке вывести итератор за конец потока (то есть когда `sbumpc()` возвращает `traits_type::eof`). У такого поведения имеются два важных следствия.

- Интервал от текущей позиции до конца потока данных определяется двумя итераторами `istreambuf_iterator<charT,traits>(поток)` (текущая позиция) и `istreambuf_iterator<charT,traits>()` (конец потока), где `поток` относится к типу `basic_istream<charT,traits>` или `basic_streambuf<charT,traits>`.
- Класс `istreambuf_iterator` не позволяет создавать подинтервалы.

Пример использования итераторов потоковых буферов

Далее представлен классический фильтр, который просто выводит все прочитанные символы при помощи итераторов потоковых буферов. Он представляет собой видоизмененную версию примера, показанного на с. 586:

```
// io/charcat2.cpp
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
    // Итератор потокового буфера ввода для cin
    istreambuf_iterator<char> inpos(cin);

    // Итератор конца потока
    istreambuf_iterator<char> endpos;

    // Итератор потокового буфера вывода для cout
    ostreambuf_iterator<char> outpos(cout);

    // Пока итератор ввода остается действительным
    while (inpos != endpos) {
        *outpos = *inpos;    // Присвоить его значение итератору вывода
        ++inpos;
        ++outpos;
    }
}
```

Юльзовательские потоковые буферы

Потоковые буфера предназначены для выполнения ввода-вывода, а их интерфейс определяется классом `basic_streambuf`. Для типов символов `char` и `wchar_t` определены специализации `streambuf` и `wstreambuf`. Эти классы используются в качестве базовых при реализации взаимодействий через специальные каналы ввода-вывода, однако для этого необходимо хорошо 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 к буферу чтения. Указатели, возвращаемые функциями `pbase()`, `pptr()` и `eptr()`, образуют интерфейс к буферу записи. Операции чтения и записи работают с этими указателями, что приводит к соответствующей реакции в канале ввода или вывода. Далее операции чтения и записи рассматриваются отдельно.

Пользовательские буферы вывода

Для буфера, используемого для записи символов, поддерживаются три указателя, которые могут быть получены функциями `pbase()`, `pptr()` и `eptr()` (рис. 13.4):

- `pbase()` («база вывода») — определяет начало буфера вывода;
- `pptr()` («указатель вывода») — определяет текущую позицию записи;
- `eptr()` («конец вывода») — определяет конец буфера вывода, то есть позицию, следующую за последним буферизуемым символом.

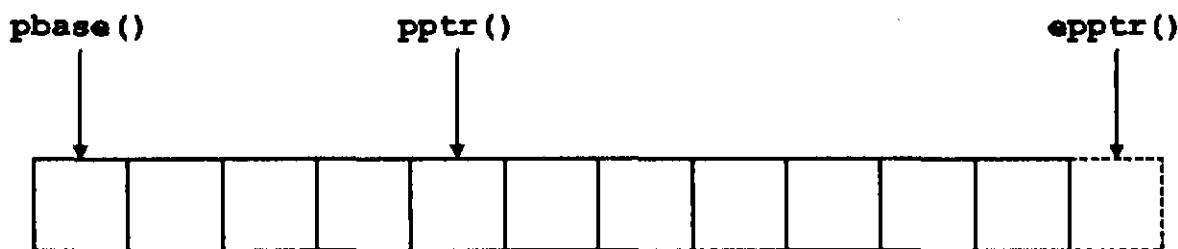


Рис. 13.4. Интерфейс буферов вывода

Символы в интервале от `pbase()` до `pptr()` (не включая символ, на который ссылается `pptr()`) уже записаны, но еще не выведены в соответствующий канал вывода.

Символы записываются в буфер функцией `sputc()`. Символ копируется в текущую позицию записи (при наличии свободной позиции), после чего указатель на текущую позицию записи увеличивается. Если буфер полон (`pptr()=eptr()`), то содержимое буфера вывода посыпается в соответствующий канал вывода вызовом виртуальной функции `overflow()`. Эта функция фактически отвечает за непосредственную передачу символов некоторому «внешнему представлению» (которое на самом деле может быть внутренним, как в случае со строковыми потоками данных). Реализация `overflow()` в базовом классе `basic_streambuf` возвращает только признак конца файла, означающий, что дальнейшая запись символов невозможна.

Функция `sputn()` позволяет записать сразу несколько символов. Она перепоручает работу виртуальной функции `xsputn()`, которая может оптимизироваться для более эффективной записи нескольких символов. Реализация `xsputn()` в классе `basic_streambuf` вызывает `sputc()` для каждого символа, поэтому в ее переопределении нет абсолютной необходимости. Тем не менее в некоторых случаях запись нескольких символов реализуется более эффективно, чем последовательная запись отдельных символов, а функция `xsputn()` помогает оптимизировать обработку символьных последовательностей.

Запись в потоковый буфер может выполняться и без буферизации — вместо этого символы выводятся сразу же после их получения. В этом случае указате-

лям буфера вывода присваивается значение 0 или NULL. Конструктор по умолчанию делает это автоматически.

На основе изложенного материала был разработан следующий пример потокового буфера, не использующего буферизацию. То есть для каждого символа вызывается функция `overflow()`. Остается лишь реализовать эту функцию.

```
// io/outbuf1.hpp
#include <streambuf>
#include <iostream>
#include <cstdio>

class outbuf : public std::streambuf
{
protected:
    /* Главная функция вывода
     * - вывод символов в верхнем регистре
     */
    virtual int_type overflow (int_type c) {
        if (c != EOF) {
            // Преобразование символа к верхнему регистру
            c = std::toupper(c,getloc());

            // Запись символа в стандартный вывод
            if (putchar(c) == EOF) {
                return EOF;
            }
        }
        return c;
    }
}:
```

В данном случае каждый символ, передаваемый в потоковый буфер, записывается функцией `putchar()` языка С. Но перед выводом символ преобразуется к верхнему регистру функцией `toupper()` (см. с. 692). Функция `getloc()` возвращает объект локального контекста, связанный с потоковым буфером (см. с. 637).

В представленном примере буфер вывода реализован специально для типа `char` (`streambuf` — специализация `basic_streambuf` для типа символов `char`). При использовании другого типа символов эту функцию следует реализовать с применением класса трактовок символов, представленного на с. 659. В этом случае сравнение `c` с концом файла выполняется иначе. Вместо `EOF` должно возвращаться значение `traits_eof()`, а если аргумент `c` равен `EOF`, следует возвращать `traits::not_eof(c)` (где `traits` — второй аргумент шаблона `basic_streambuf`). Возможная реализация выглядит так:

```
// io/outbuf1x.hpp
#include <streambuf>
#include <iostream>
#include <cstdio>

template <class charT, class traits = std::char_traits<charT> >
class basic_outbuf : public std::basic_streambuf<charT,traits>
```

```

{
protected:
/* Главная функция вывода
 * - вывод символов в верхнем регистре
 */
virtual typename traits::int_type overflow (typename traits::int_type c) {
    if (!traits::eq_int_type(c,traits::eof())) {
        // Преобразование символа к верхнему регистру
        c = std::toupper(c.getloc());

        // Запись символа в стандартный вывод
        if (putchar(c) == EOF) {
            return traits::eof();
        }
    }
    return traits::not_eof(c);
}
};

typedef basic_outbuf<char>    outbuf;
typedef basic_outbuf<wchar_t> woutbuf;

```

Пусть этот потоковый буфер используется в представленной ниже программе:

```

// io/outbuf1.cpp
#include <iostream>
#include "outbuf1.hpp"

int main()
{
    outbuf ob;           // Создание специального буфера вывода
    std::ostream out(&ob); // Инициализация выходного потока
                          // созданным буфером вывода
    out << "31 hexadeciml: " << std::hex << 31 << std::endl;
}

```

В этом случае будет получен следующий результат:

31 HEXADECIMAL: 1F

Аналогичный подход может использоваться при записи в другие приемники. Так, для инициализации объекта конструктору потокового буфера можно передать дескриптор файла, имя сокетного соединения или два других потоковых буфера, используемые для одновременной записи. Чтобы организовать вывод в соответствующий приемник, достаточно реализовать функцию `overflow()`. Кроме того, функцию `xspitn()` следует реализовать для оптимизации вывода в потоковый буфер.

Чтобы конструировать потоковые буфера было проще, рекомендуется 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()` (низкоуровневая функция ввода-вывода в системах семейства UNIX). Также определяется класс, производный от `ostream`, — он поддерживает потоковый буфер, которому передается файловый дескриптор.

```
// io/oubuf2.hpp
#include <iostream>
#include <streambuf>
#include <cstdio>

extern "C" {
    int write (int fd, const char* buf, int num);
}

class fdoutbuf : public std::streambuf {
protected:
    int fd; // Файловый дескриптор
public:
    // Конструктор
    fdoutbuf (int _fd) : fd(_fd) {
    }
protected:
    // Запись одного символа
    virtual int_type overflow (int_type c) {
        if (c != EOF) {
            char z = c;
            if (write (fd, &z, 1) != 1) {
                return EOF;
            }
        }
        return c;
    }
    // Запись нескольких 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);
    }
};
```

Потоковый буфер также реализует функцию `xsputn()`, позволяющую предотвратить вызов `overflow()` для каждого символа при отправке в буфер последовательности символов. Функция записывает всю последовательность символов в файл, заданный дескриптором `fd`, за один вызов. Функция `xsputn()` возвращает количество успешно выведенных символов. Пример:

```
// io/outbuf2.cpp
#include <iostream>
#include "outbuf2.hpp"

int main()
{
    fdostream out(1); // Поток с буфером, записывающим по дескриптору 1
    out << "31 hexadecimal: " << std::hex << 31 << std::endl;
}
```

Программа создает выходной поток данных, инициализируемый дескриптором файла 1. По действующим правилам этот дескриптор соответствует стандартному каналу вывода. Следовательно, в данном примере символы будут просто направляться в стандартный выходной поток данных. В аргументе конструктора также могут использоваться другие дескрипторы (например, дескриптор файла или сокета).

Чтобы потоковый буфер действительно обеспечивал буферизацию, буфер вывода должен быть инициализирован функцией `setp()`. В следующем примере показано, как это делается:

```
// io/outbuf3.hpp
#include <cstdio>
#include <streambuf>

extern "C" {
    int write (int fd, const char* buf, int num);
}

class outbuf : public std::streambuf {
protected:
    static const int bufferSize = 10; // Размер буфера данных
    char buffer[bufferSize]; // Буфер данных

public:
    /* Конструктор
     * - Инициализация 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:
// Вывод символов, хранящихся в буфере
int flushBuffer () {
    int num = pptr()-pbase();
    if (write (1, buffer, num) != num) {
        return EOF;
    }
    pbump (-num); // Соответствующий перевод указателя вывода
    return num;
}

/* Буфер полон
 * - Вывести с и все предшествующие символы
 */
virtual int_type overflow (int_type c) {
    if (c != EOF) {
        // Вставка символа в буфер
        *pptr() = c;
        pbump(1);
    }
    // Очистка буфера
    if (flushBuffer() == EOF) {
        // ERROR
        return EOF;
    }
    return c;
}

/* Синхронизация данных с файлом/приемником
 * - вывод данных из буфера
 */
virtual int sync () {
    if (flushBuffer() == EOF) {
        // ОШИБКА
        return -1;
    }
    return 0;
}
};

Конструктор инициализирует буфер вывода функцией setp():
```

```
setp (buffer, buffer+(bufferSize-1));
```

Буфер вывода настроен так, что функция `overflow()` вызывается, когда еще остается место для одного символа. Если функция `overflow()` вызывается с аргументом, отличным от `EOF`, то соответствующий символ может быть помещен в текущую позицию записи, поскольку указатель на нее не выходит за пределы конечного указателя. После того как аргумент `overflow()` будет помещен в позицию записи, буфер можно очистить.

Именно эта задача решается функцией `flushBuffer()`. Функция записывает символы в стандартный канал вывода (дескриптор 1) при помощи функции `write()`. Функция `rdbuf()` потокового буфера возвращает позицию записи к началу буфера.

Функция `overflow()` вставляет в буфер символ, ставший причиной вызова `overflow()`, если он отличен от `EOF`. Затем функция `rdbuf()` смешает текущую позицию записи, чтобы она отражала новый конец блока буферизованных символов. Тем самым позиция записи временно смещается за конечную позицию (`egptr()`).

Класс также содержит виртуальную функцию `sync()`, предназначенную для синхронизации текущего состояния потокового буфера с непосредственным носителем данных. Обычно синхронизация ограничивается простой очисткой буфера. Для небуферизованных версий переопределять эту функцию не обязательно, поскольку нет самого буфера, который можно было бы очистить.

Виртуальный деструктор обеспечивает вывод данных, остающихся в буфере при его уничтожении.

Эти функции переопределяются для большинства потоковых буферов. Если внешнее представление имеет особую структуру, возможно, придется переопределить дополнительные функции, например функции `seekoff()` и `seekpos()`, для управления позицией записи.

Пользовательские буфера ввода

В сущности, механизм ввода работает по тем же принципам, что и механизм вывода. Однако для ввода также существует возможность отмены последнего чтения. Функции `sungetc()` (вызывается функцией `unget()` входного потока данных) и `sputbackc()` (вызывается функцией `putback()` входного потока данных) используются для восстановления потокового буфера в состоянии перед последним чтением. Также 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.

Для буфера, используемого для записи символов, поддерживаются три указателя, которые могут быть получены функциями `eback()`, `gptr()` и `egptr()` (рис. 13.5):

- `eback()` («база ввода») — определяет начало буфера ввода или конец области отката;
- `gptr()` («указатель ввода») — определяет текущую позицию чтения;
- `egptr()` («конец ввода») — определяет конец буфера ввода.

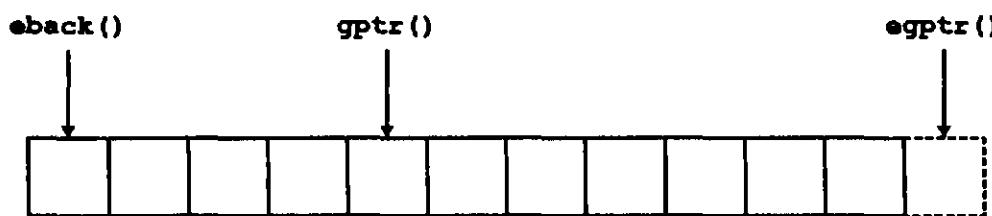


Рис. 13.5. Интерфейс чтения из потоковых буферов

Символы, находящиеся между начальной и конечной позициями, были переданы из внешнего представления в память программы, но еще ожидают обработки.

Одиночные символы читаются функциями `sgetc()` и `sbumpc()`. Отличие между этими функциями состоит в том, что функция `sbumpc()` увеличивает указатель текущей позиции ввода, а функция `sgetc()` этого не делает. Если буфер будет полностью прочитан (`gptr() == egptr()`), значит, доступных символов нет и буфер необходимо заполнять заново. Для этого вызывается виртуальная функция `underflow()`, отвечающая за чтение данных. С другой стороны, функция `sbumpc()` при отсутствии символов вызывает виртуальную функцию `uflow()`. По умолчанию `uflow()` просто вызывает `underflow()`, а затем увеличивает указатель. По умолчанию версия `underflow()` в базовом классе `basic_streambuf` возвращает `EOF`, то есть признак невозможности дальнейшего чтения с использованием стандартной реализации.

Функция `sgetn()` предназначена для чтения сразу нескольких символов. Она перепоручает работу виртуальной функции `xsgetn()`. В реализации по умолчанию `xsgetn()` просто читает символы, вызывая для каждого из них `sbumpc()`. По аналогии с функцией `xspurtn()` при записи функция `xsgetn()` используется для оптимизации чтения нескольких символов.

В отличие от вывода для ввода недостаточно переопределить одну функцию. Вам придется либо выполнить настройку буфера, либо, по крайней мере, реализовать функции `underflow()` и `uflow()`. Дело в том, что функция `underflow()` не перемещается за текущий символ, однако она может быть вызвана из `sgetc()`. Перемещение к следующему символу приходится выполнять путем манипуляций с буфером или вызовом `uflow()`. В любом случае функция `underflow()` должна быть реализована для любого потокового буфера, поддерживающего чтение символов. Если реализованы обе функции, `underflow()` и `uflow()`, в настройке буфера нет необходимости.

Настройка буфера чтения осуществляется функцией `setg()`, получающей следующие три аргумента (именно в таком порядке):

- указатель на начало буфера (`eback()`);
- указатель на текущую позицию чтения (`gptr()`);
- указатель на конец буфера (`egptr()`).

В отличие от `setp()` функция `setg()` вызывается с тремя аргументами. Это необходимо для того, чтобы вы могли зарезервировать память для символов, возвращаемых в поток данных. Таким образом, при настройке буфера ввода желательно, чтобы некоторые символы (по крайней мере один) уже были прочитаны, но еще не сохранены в буфере.

Как упоминалось выше, символы можно вернуть в буфер чтения с помощью функций `sputbackc()` и `sungetc()`. Функция `sputbackc()` получает возвращаемый символ в аргументе и проверяет, что именно этот символ был прочитан последним. Обе функции уменьшают указатель текущей позиции чтения, если это возможно. Очевидно, это возможно только в том случае, если указатель чтения не находится в начале буфера. При попытке возврата символа в начале 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, следует реализовать функцию `pbackfail()`, потому что в общем случае предполагается, что хотя бы один символ может быть возвращен в поток.

Когда буфер заполняется заново, возникает другая проблема: если прежние данные не были сохранены в буфере, возврат даже одного символа невозможен. По этой причине реализация `underflow()` часто перемещает несколько последних символов (например, четыре) в начало буфера и присоединяет читаемые символы после них. Это позволяет вернуть хотя бы несколько символов перед тем, как будет вызвана функция `pbackfail()`.

Следующий пример показывает, как может выглядеть подобная реализация. Класс `inbuf` реализует буфер ввода, рассчитанный на десять символов. Буфер условно делится на две части: «область возврата» из четырех символов и «нормальный» буфер ввода из шести символов.

```
// io/inbuf1.hpp
#include <cstdio>
#include <cstring>
#include <streambuf>

extern "C" {
    int read (int fd, char* buf, int num);
}

class inbuf : public std::streambuf {
protected:
    /* Буфер данных:
     * - до четырех символов в области возврата.
     * - до шести символов в обычном буфере ввода.
     */
    static const int bufferSize = 10;      // Размер буфера данных
    char buffer[bufferSize];              // Буфер

public:
    /* Конструктор
     * - Инициализация пустого буфера
     * - без области возврата
     * => принудительный вызов underflow()*
     */
}
```

```
inbuf() {
    setg (buffer+4,           // Начало области возврата
          buffer+4,           // Текущая позиция
          buffer+4);          // Конечная позиция
}

protected:
    // Вставка новых символов в буфер
    virtual int_type underflow () {

        // Текущая позиция чтения предшествует концу буфера?
        if (gptr() < egptr()) {
            return traits_type::to_int_type(*gptr());
        }

        /* Обработка разиера области возврата
         * - использовать количество прочитанных символов,
         * - но не более 4
         */
        int numPutback;
        numPutback = gptr() - eback();
        if (numPutback > 4) {
            numPutback = 4;
        }

        /* Копирование до четырех ранее прочитанных символов
         * в область возврата (первые четыре символа)
         */
        std::memmove (buffer+(4-numPutback), gptr()-numPutback,
                      numPutback);

        // Чтение новых символов
        int num;
        num = read (0, buffer+4, bufferSize-4);
        if (num <= 0) {
            // ОШИБКА или EOF
            return EOF;
        }

        // Сброс указателей
        setg (buffer+(4-numPutback),      // Начало области возврата
              buffer+4,                  // Текущая позиция чтения
              buffer+4+num);             // Конец буфера

        // Вернуть следующий символ
        return traits_type::to_int_type(*gptr());
    }
};
```

Конструктор инициализирует все указатели так, что буфер остается абсолютно пустым (рис. 13.6). При попытке прочитать символы из этого буфера вызывается функция `underflow()`, всегда используемая потоковыми буферами для чтения следующих символов. Сначала функция проверяет наличие прочитанных символов в буфере ввода. Если такие символы есть, они перемещаются в область возврата функцией `tempstr()`. 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 изменившейся ситуацией, возвращается первый прочитанный символ.

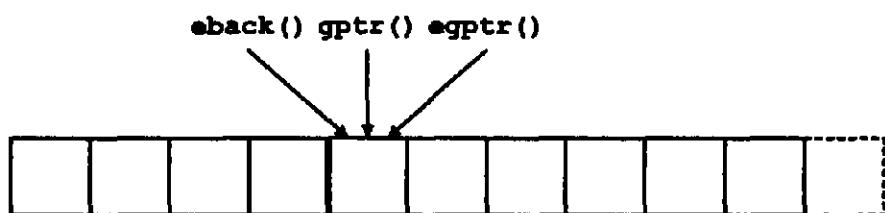


Рис. 13.6. Буфер ввода после инициализации

Предположим, при первом вызове `read()` были прочитаны символы `H, a, l, l, o, w`. Буфер ввода переходит в состояние, изображенное на рис. 13.7. Область возврата пуста, потому что буфер был заполнен в первый раз, и готовых к возврату символов еще нет.

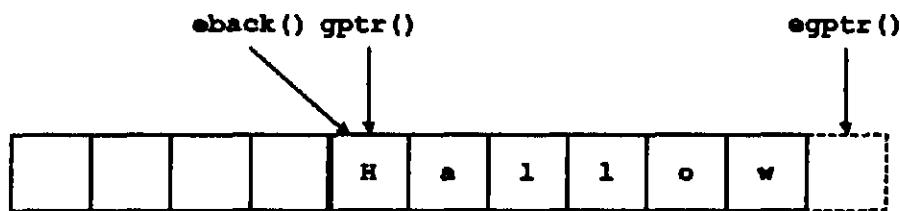


Рис. 13.7. Буфер ввода после чтения символов `H a l l o w`

После извлечения этих символов последние четыре символа перемещаются в область возврата, после чего читаются новые символы. Допустим, при следующем вызове `read()` были прочитаны символы `e, e, n, \n` (рис. 13.8).

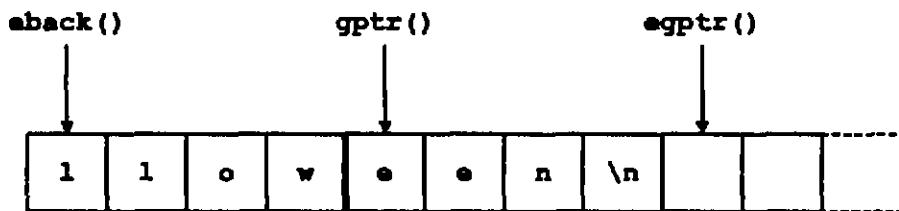


Рис. 13.8. Буфер ввода после чтения еще четырех символов

Пример использования этого потокового буфера:

```
// io/unbuf1.cpp
#include <iostream>
```

```
#include "inbuf1.hpp"

int main()
{
    inbuf ib;           // Создание специального потокового буфера
    std::istream in(&ib); // Инициализация выходного потока этим буфером

    char c;
    for (int i=1; i<=20; i++) {
        // Чтение следующего символа (из буфера)
        in.get(c);

        // Вывод символа и очистка буфера
        std::cout << c << std::flush;

        // После вывода восьми символов
        // вернуть два последних символа в поток
        if (i == 8) {
            in.unget();
            in.unget();
        }
    }
    std::cout << std::endl;
}
```

Программа в цикле читает символы и выводит их в поток данных `cout`. После чтения восьмого символа выполняется возврат на два символа, в результате чего седьмой и восьмой символы выводятся дважды.

Проблемы эффективности

Данный раздел посвящен вопросам эффективности. Вообще говоря, потоковые классы обычно работают достаточно эффективно, но в приложениях, критичных по быстродействию ввода-вывода, их можно сделать еще эффективнее.

Одна из проблем быстродействия уже упоминалась на с. 567: 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.

Синхронизация со стандартными потоками данных 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 `clog` и `wclog`

используют тот же потоковый буфер, что и `cerr` и `wcerr` соответственно. Таким образом, по умолчанию они синхронизируются с `stderr`, хотя в стандартной библиотеке С у этих потоков данных нет прямых аналогов.

В зависимости от реализации синхронизация может приводить к лишним затратам. Например, если стандартные потоки данных C++ реализованы с использованием стандартных файлов С, это фактически подавляет буферизацию соответствующих потоковых буферов. Однако буферизация необходима для выполнения некоторых оптимизаций, особенно для форматированного чтения (см. далее). Чтобы программист мог переключиться на нужную реализацию, в классе `ios_base` определена статическая функция `sync_with_stdio()` (табл. 13.46).

Таблица 13.46. Синхронизация стандартных потоков данных C++ и С

Статическая функция	Описание
<code>sync_with_stdio()</code>	Возвращает информацию о том, синхронизируются ли стандартные объекты потоков данных со стандартными потоками данных С
<code>sync_with_stdio(false)</code>	Запрещает синхронизацию потоков данных C++ и С (при условии, что функция была вызвана до первой операции ввода-вывода)

При вызове функции `sync_with_stdio()` передается необязательный логический аргумент, который указывает, нужно ли активизировать синхронизацию со стандартными потоками данных С. Для отключения синхронизации функция вызывается с аргументом `false`:

```
std::ios::sync_with_stdio(false); // Отключение синхронизации
```

Помните, что синхронизация отключается только до выполнения любой 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 `true`, отражающее состояние по умолчанию для стандартных потоков данных.

Буферизация в потоковых буферах

Буферизация ввода-вывода также является важным фактором эффективности. Системные вызовы обычно обходятся относительно дорого, поэтому их количество должно быть по возможности сведено к минимуму. Тем не менее существует и другая, более тонкая причина для буферизации в потоковых буферах C++ (по крайней мере, при вводе): функции форматного ввода-вывода работают с потоками данных при помощи итераторов потоковых буферов, а операции с итераторами медленнее операций с указателями. Отличия не так уж велики, но вполне достаточны для того, чтобы оправдать применение оптимизированных реализаций для частых операций (например, форматированного ввода числовых данных). Однако для этого необходимо применение буферизации в потоковых буферах.

Итак, весь ввод-вывод осуществляется через потоковые буферы, обеспечивающие механизм буферизации. Но полагаться только на эту буферизацию недостаточно по трем причинам.

- Потоки данных без буферизации часто реализуются проще. Если соответствующие потоки данных используются редко или только для вывода (для вывода различия между итераторами и указателями не столь существенны, как для ввода; основная проблема — сравнение итераторов потоковых буферов), вероятно, буферизация не играет особой роли. Но если потоковый буфер интенсивно используется, для него определенно следует реализовать буферизацию.
- При установленном флаге `unbuf` выходной поток данных очищает буфер после каждой операции вывода. Кроме того, очистка производится манипуляторами `flush` и `endl`. Вероятно, для оптимального быстродействия следует избегать всех трех способов. Но при выводе на консоль, например, было бы логично очищать буфер после вывода полных строк. Если вы зашли в тупик с программой, интенсивно использующей манипуляторы `unbuf`, `flush` и `endl`, рассмотрите возможность применения специального потокового буфера, который в соответствующий момент вызывает не функцию `sync()`, а другую функцию.
- Связывание потоков данных функцией `tie()` (см. с. 612) также требует дополнительных операций очистки потоков данных. Следовательно, связывание должно применяться только при абсолютной необходимости.

При разработке новых потоковых буферов рекомендуется сначала реализовать их без буферизации. Если потоковый буфер окажется «узким местом» в работе системы, вы сможете организовать буферизацию так, чтобы не затронуть другие компоненты приложения.

Непосредственная работа с потоковыми буферами

Все функции классов `basic_istream` и `basic_ostream`, выполняющие чтение или запись символов, работают по одной схеме: сначала конструируется соответствующий объект `sentry`, а затем выполняется операция. Конструирование объекта `sentry` приводит к очистке буферов возможных связанных объектов, игнорированию пропусков (только при вводе) и выполнению операций, специфических для конкретных реализаций, например операций блокировки файлов в средах с параллельным функционированием нескольких потоков выполнения (`threads`), то есть в многопоточных средах (см. с. 631).

При неформатированном вводе-выводе многие операции потоков данных все равно бесполезны, разве что операция блокировки может пригодиться при работе с потоками в средах с параллельным функционированием нескольких потоков выполнения (поскольку в C++ проблемы многопоточности не решаются). Следовательно, при неформатированном вводе-выводе прямая работа с потоковыми буферами обычно более эффективна.

Для этого можно определить для потоковых буферов операторы << и >>.

- При получении указателя на потоковый буфер оператор << выводит все накопленные данные. Вероятно, это самый быстрый способ копирования файлов с использованием потоков данных C++. Пример:

```
// io/copy1.cpp
#include <iostream>

int main ()
{
    // Копирование стандартного ввода в стандартный вывод
    std::cout << std::cin.rdbuf();
}
```

В этом фрагменте функция `rdbuf()` возвращает буфер `cin` (см. с. 613). Следовательно, программа копирует весь стандартный входной поток данных в стандартный выходной поток.

- При получении указателя на потоковый буфер оператор >> выполняет прямое чтение данных. Например, копирование стандартного входного потока данных в стандартный выходной поток также может выполняться следующим образом:

```
// io/copy2.cpp
#include <iostream>

int main ()
{
    // Копирование стандартного ввода в стандартный вывод
    std::cin >> std::noskipws >> std::cout.rdbuf();
}
```

Обратите внимание наброс флага `skipws`. В противном случае будут проигнорированы начальные пропуски во входных данных (см. с. 600).

Впрочем, прямая работа с потоковым буфером может быть оправданна 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. Внутри цикла пропуски игнорируются вручную (использование манипулятора `ws` также привело бы к конструированию объекта `sentry`), а прямое чтение числовых данных осуществляется фасетом `num_get` (см. с. 677).

Учтите, что потоковый буфер не обладает собственным состоянием ошибки. Он также ничего не знает о входных или выходных потоках, которые могут к нему подключиться. Следовательно, внутри следующего фрагмента состояние ошибки `in` не может измениться из-за сбоя или достижения конца файла:

```
// Копирование содержимого in в out
out << in.rdbuf();
```

14 Интернационализация

С развитием глобального рынка *интернационализация* стала играть более важную роль в разработке программного обеспечения. По этой причине в стандартную библиотеку C++ были включены средства написания локализованного кода. В основном они связаны с вводом-выводом и обработкой строк. Этим средствам и посвящена данная глава. Большое спасибо Дитмару Кюлю (Dietmar Kühl), эксперту в области ввода-вывода и средств интернационализации стандартной библиотеки C++, — он написал большую часть этой главы.

Стандартная библиотека C++ предоставляет общие средства поддержки национальных стандартов без привязки к конкретным системам и правилам. Например, строки не ограничиваются конкретным типом символов, чтобы обеспечить поддержку 16-разрядных азиатских кодировок символов. При интернационализации программ должны учитываться следующие обстоятельства.

- Разные кодировки символов обладают разными свойствами. Для работы с ними необходимы гибкие решения практических вопросов, например, что считать буквой или, еще хуже, какой тип должен использоваться для представления символов. Тип `char` не подходит для кодировок, содержащих более 256 символов.
- Пользователь рассчитывает, что применяемая им программа соответствует национальным и культурным стандартам (например, при форматировании денежных величин, чисел и логических значений).

В обоих случаях стандартная библиотека C++ предоставляет соответствующие средства.

Основной механизм интернационализации основан на использовании *объекта локального контекста*. Локальный контекст (*locale*) отражает расширяемый набор правил, адаптируемых для конкретных национальных стандартов. Локальные контексты уже применялись в языке C. В стандарте C++ этот механизм был обобщен и стал более гибким. Механизм локальных контекстов 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, стандартный размер бумаги и т. д.

Многие механизмы интернационализации практически не требуют дополнительной работы со стороны программиста. Например, в потоковом механизме ввода-вывода C++ числовые данные форматируются по правилам локального контекста. Программисту остается лишь позаботиться о том, чтобы классы потоков данных для ввода-вывода учитывали предпочтения пользователя.

Наряду с автоматическим использованием программист может напрямую обращаться к объектам локального контекста для форматирования, классификации символов и т. д. Некоторые средства интернационализации, поддерживаемые стандартной библиотекой C++, не требуются самой библиотеке — программисты должны вызывать эти функции в своих программах. Например, в стандартной библиотеке C++ не определены потоковые функции для форматирования времени, даты или денежных величин. Чтобы выполнить такое форматирование, необходимо вызвать соответствующую функцию (например, из пользовательского потокового оператора для вывода объектов денежных величин).

Интернационализация строк и потоков данных основана на концепции *трактовок символов*. Трактовки определяют базовые свойства и операции, зависящие от кодировки (такие, как признак «конца файла» или функции сравнения, присваивания и копирования строк).

Классы интернационализации были включены в стандарт относительно поздно. Хотя общий механизм чрезвычайно гибок, он нуждается в некоторой доработке. Например, функции контекстного сравнения строк (то есть сравнения строк с учетом правил локального контекста) используют только итераторы типа `const charT*`, где `charT*` — некоторый тип символов. Хотя с очень большой вероятностью в классе `basic_string<charT>` задействуется именно этот тип итератора, это вовсе не гарантировано. Следовательно, не гарантировано и то, что строковые итераторы могут передаваться в аргументах функций контекстного сравнения строк. С другой стороны, результат вызова функции `data()` класса `basic_string` может передаваться функциям контекстного сравнения.

Различия в кодировках символов

Одна из проблем, решаемых средствами интернационализации — поддержка разных кодировок символов. Эта проблема характерна в основном для Азии, где для представления символов используются разные кодировки. Как правило, в таких кодировках для кодирования каждого символа приходится задействовать более 8 бит, поэтому обработка текста требует новых концепций и функций.

Расширенные и многобайтовые кодировки

Существуют два основных принципа определения кодировок, содержащих более 256 символов: *многобайтовое* и *расширенное* представления.

- В *многобайтовых* кодировках символы представляются переменным количеством байтов. За однобайтовым символом (например, символом из кодировки ISO Latin-1) может следовать трехбайтовый символ (японский иероглиф).

- В *расширенных* кодировках символ всегда представляется постоянным количеством байтов независимо от его типа. В типичных кодировках символ представляется величиной 2 или 4 байта. На концептуальном уровне такие кодировки не отличаются от представлений, в которых для локальных контекстов, где достаточно кодировки ISO Latin-1 или даже ASCII, используется однобайтовая кодировка.

Многобайтовое представление более компактно по сравнению с расширенным, поэтому для хранения данных вне программ обычно применяется многобайтовое представление. И наоборот, с символами фиксированного размера гораздо удобнее работать, поэтому в программах обычно используется расширенное представление.

В ISO C++, как и в ISO C, используется тип `wchar_t` для расширенных кодировок. С другой стороны, в C++ `wchar_t` является ключевым словом, а не определением типа, что позволяет перегружать все функции с этим типом.

В многобайтовой строке один и тот же байт может представлять целый символ или его часть. В процессе перебора содержимого многобайтовой строки каждый байт интерпретируется согласно текущему «состоянию сдвига». В зависимости от значения байта и текущего состояния сдвига байт может представлять символ или изменение состояния сдвига. Многобайтовая строка всегда начинается с определенного исходного состояния сдвига. Например, в исходном состоянии байты могут представлять символы ISO Latin-1 до тех пор, пока не будет обнаружен специальный символ перехода. Символ, следующий за ним, определяет новое состояние сдвига. Допустим, в новом состоянии сдвига байты могут интерпретироваться как арабские символы до тех пор, пока не будет обнаружен следующий символ перехода.

Преобразование между кодировками символов осуществляется при помощи шаблона `codecvt<>` (см. с. 689). Этот класс используется в основном классом `basic_filebuf<>` (см. с. 602) для преобразования между внутренними и внешними представлениями. В стандарте C++ многобайтовые кодировки символов не оговариваются, но в нем предусмотрена запись состояния сдвига. У функций класса `codecvt<>` имеется аргумент, в котором может храниться произвольное состояние строки. Кроме того, класс поддерживает функцию для определения последовательности символов, используемой для возврата к исходному состоянию сдвига.

ТРАКТОВКИ СИМВОЛОВ

Различия в кодировках существенны для обработки строк и ввода-вывода. Например, признак «конца файла» и конкретные особенности сравнения символов могут различаться в зависимости от реализации.

Предполагается, что строковые и потоковые классы будут специализироваться встроенными типами, прежде всего `char` и `wchar_t`. Интерфейс встроенных типов должен оставаться неизменным, поэтому информация о разных аспектах представления символов выделяется в отдельный класс — так называемый *класс трактовок символов*. Он передается строковым и потоковым классам в аргументе

шаблона. По умолчанию в этом аргументе передается класс `char_traits`, параметризованный по аргументу, определяющему тип символов строки или потока данных:

```
namespace std {
    template<class charT,
              class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
    class basic_string;
}

namespace std {
    template<class charT,
              class traits = char_traits<charT> >
    class basic_istream;
    template<class charT,
              class traits = char_traits<charT> >
    class basic_ostream;
    ...
}
```

Трактовки символов указываются в классе `char_traits<>`, который определяется в заголовочном файле `<string>` и параметризуется по конкретному типу символов:

```
namespace std {
    template <class charT>
    struct char_traits {
        ...
    };
}
```

Классы трактовок определяют все основные свойства типа символов и операции, необходимые для реализации строк и потоков данных как статических компонентов. Члены класса `char_traits` перечислены в табл. 14.1.

Таблица 14.1. Члены класса трактовок символов

Выражение	Описание
<code>char_type</code>	Тип символов (то есть аргумент шаблона <code>char_traits</code>)
<code>int_type</code>	Тип, размеры которого достаточны для представления дополнительного, не используемого для других целей признака конца файла
<code>pos_type</code>	Тип, используемый для представления позиций в потоке
<code>off_type</code>	Тип, используемый для представления смещений между позициями в потоке
<code>state_type</code>	Тип, используемый для представления текущего состояния в многобайтовых потоках
<code>assign(c1, c2)</code>	Присваивает <code>c1</code> символ <code>c2</code>
<code>eq(c1, c2)</code>	Проверяет равенство символов <code>c1</code> и <code>c2</code>

Выражение	Описание
<code>lt(c1, c2)</code>	Проверяет условие «символ <code>c1</code> меньше символа <code>c2</code> »
<code>length(s)</code>	Возвращает длину строки <code>s</code>
<code>compare(s1, s2, n)</code>	Сравнивает до <code>n</code> символов строк <code>s1</code> и <code>s2</code>
<code>copy(s1, s2, n)</code>	Копирует <code>n</code> символов строки <code>s2</code> в <code>s1</code>
<code>move(s1, s2, n)</code>	Копирует <code>n</code> символов строки <code>s2</code> в <code>s1</code> , причем строки <code>s1</code> и <code>s2</code> могут перекрываться
<code>assign(s, n, c)</code>	Присваивает символ <code>c</code> <code>n</code> символам строки <code>s</code>
<code>find(s, n, c)</code>	Возвращает указатель на первый символ строки <code>s</code> , равный <code>c</code> ; если среди первых <code>n</code> символов такой символ отсутствует, возвращает 0
<code>eof()</code>	Возвращает признак конца файла
<code>to_int_type(c)</code>	Преобразует символ <code>c</code> в соответствующее представление типа <code>int_type</code>
<code>to_char_type(i)</code>	Преобразует представление <code>i</code> типа <code>int_type</code> в символ (результат преобразования EOF не определен)
<code>not_eof(i)</code>	Возвращает значение <code>i</code> , если <code>i</code> не является представлением EOF; в этом случае возвращается значение, определяемое реализацией (и отличное от EOF)
<code>eq_int_type(i1, i2)</code>	Проверяет равенство двух символов <code>i1</code> и <code>i2</code> , представленных в виде типа <code>int_type</code> (иначе говоря, аргументы могут быть равны EOF)

Функции обработки строк или последовательностей символов присутствуют только по соображениям оптимизации. Они могут быть реализованы при помощи функций, обрабатывающих отдельные символы. Например, функция `copy()` реализуется на базе `assign()`, однако при работе со строками возможны более эффективные реализации.

Все значения количества символов в табл. 14.1 задаются точно, то есть без учета символов завершения строк.

Последняя группа функций предназначена для работы с признаком конца файла (EOF). Этот служебный символ расширяет кодировку и требует специальной обработки. Для некоторых представлений типа символов может оказаться недостаточно, поскольку значение символа EOF должно отличаться от значений всех «обычных» символов кодировки. По правилам языка С функции чтения символов возвращали тип `int` вместо `char`. В C++ эта методика была усовершенствована. В трактовках символов `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. Для некоторых классов трактовок типы `char_type` и `int_type` могут быть идентичными. Например, если не все значения типа `char_type` необходимы для представления символов, одно из свободных значений может использоваться для представления символа конца файла.

Типы `pos_type` и `off_type` используются для определения позиций и смещений (подробности см. на с. 609).

В стандартную библиотеку C++ включены специализации `char_traits<>` для типов `char` и `wchar_t`:

```
namespace std {
    template<> struct char_traits<char>;
    template<> struct char_traits<wchar_t>;
}
```

Специализация для `char` обычно реализуется при помощи глобальных строковых функций языка C, определяемых в файлах `<cstring>` и `<string.h>`. Примерная реализация может выглядеть так:

```
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 void 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* s1, 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)(insigned 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 трактовок, обеспечивающего работу со строками без учета регистра символов, приведена на с. 485.

Интернационализация специальных символов

При рассказе об интернационализации остался без ответа один важный вопрос: как интернационализируются специальные символы (например, символ \n)? Класс `basic_ios` содержит функции `widen()` и `narrow()`, которые могут использоваться для этой цели. Например, символ новой строки в кодировке, соответствующей формату потока данных `strm`, может записываться следующим образом:

```
strm.widen('\n'); // Интернационализированный символ новой строки
```

Символ конца строки в этой же кодировке выглядит так:

```
strm.widen('\0'); // Интернационализированный завершитель строк
```

За примером обращайтесь к реализации манипулятора `endl` на с. 587.

Функции `widen()` и `narrow()` используют объект локального контекста, а говоря точнее – фаэт `ctype` этого объекта. Фаэт может потребоваться для преобразования символов между `char` и другим представлением (см. с. 689). Например, следующее выражение преобразует символ с типа `char` в объект типа `char_type` с использованием объекта локального контекста `loc`¹:

```
std::use_facet<std::ctype<char_type>>(loc).widen(c)
```

Подробности работы с локальными контекстами и их фаэтами будут представлены далее.

¹ Обратите внимание на пробел между символами `>`. Последовательность `>` воспринимается компилятором как оператор сдвига, что приводит к синтаксической ошибке.

Концепция локального контекста

Распростраиненный подход к интернационализации основан на использовании специальных сред, называемых *локальными контекстами* (*locale*) и инкапсулирующих национальные или культурные стандарты. Именно этот подход используется в языке С. Таким образом, в отношении интернационализации локальный контекст представляет собой набор параметров и функций, обеспечивающих поддержку национальных или культурных стандартов. В соответствии с конвенциями X/Open¹ локальный контекст используется переменной окружения с именем *LANG*. В зависимости от локального контекста выбираются разные форматы вещественных чисел, дат, денежных сумм и т. д.

Обычно локальный контекст определяется строкой в формате

`язык[_зона[.код]]`

Здесь *язык* — обозначение языка (например, английский или немецкий), а *зона* — страна, географический регион или культура, в которой используется этот язык. В частности, этот квалификатор позволяет поддерживать национальные стандарты даже в том случае, если на одном языке говорят в разных странах. Квалификатор *код* определяет кодировку символов. Он важен, прежде всего, для азиатских стран, где один набор символов может существовать в нескольких кодировках.

В табл. 14.2 приведена подборка типичных определений локальных контекстов. Однако следует помнить, что эти строки еще не стандартизированы (так, первый символ языка может записываться в верхнем регистре). Некоторые реализации отличаются от представленного формата; например, английский локальный контекст в них обозначается строкой *english*. В конечном счете поддержка тех или иных локальных контекстов системой зависит от реализации.

Таблица 14.2. Примеры имен локальных контекстов

Имя	Описание
C	Используется по умолчанию: соглашения стандарта ANSI-C (английский, 7-разрядная кодировка)
de_DE	Немецкий язык (Германия)
de_DE.88591	Немецкий язык (Германия) с кодировкой ISO Latin-1
de_AT	Немецкий язык (Австрия)
de_CH	Немецкий язык (Швейцария)
en_US	Английский язык (США)
en_GB	Английский язык (Великобритания)
en_AU	Английский язык (Австралия)
en_CA	Английский язык (Канада)
fr_FR	Французский язык (Франция)

¹ POSIX и X/Open — стандартные спецификации интерфейсов операционных систем.

Имя	Описание
fr_CH	Французский язык (Швейцария)
fr_CA	Французский язык (Канада)
ja_JP.jis	Японский язык (Япония) с кодировкой JIS (Japanese Industrial Standard)
ja_JP.sjis	Японский язык (Япония) с кодировкой Shift JIS
ja_JP.ujis	Японский язык (Япония) с кодировкой UNIXized JIS
ja_JP.EUC	Японский язык (Япония) с кодировкой EUC (Extended Unix Code)
ko_kr	Корейский язык (Корея)
zh_CN	Китайский язык (Китай)
zh_TW	Китайский язык (Тайвань)
lt_LN.bit7	ISO Latin (7-разрядная кодировка)
lt_LN.bit8	ISO Latin (8-разрядная кодировка)
POSIX	Соглашения стандарта POSIX: английский, 7-разрядная кодировка

Несмотря на обилие имен локальных контекстов, у программ обычно не возникает особых проблем с нестандартными именами! Дело в том, что информация локального контекста в той или иной форме предоставляется пользователем. Как правило, программа просто читает информацию из переменных окружения или базы данных и определяет, с какими локальными контекстами она должна работать. Таким образом, хлопоты с определением правильных имен контекстов возлагаются на пользователя. Только если программа всегда использует конкретный локальный контекст, имя этого контекста жестко кодируется в программе. Обычно в таких случаях достаточно локального контекста C, который заведомо поддерживается всеми реализациями под именем C.

Далее показано, как работать с разными локальными контекстами в программе на C++. Кроме того, описываются *фацеты* локальных контекстов, позволяющие задействовать конкретные варианты форматирования.

В языке C предусмотрен также механизм применения кодировок, содержащих более 256 символов. Этот механизм основан на использовании символов `wchar_t` – синонима для одного из целочисленных типов с языковой поддержкой констант и строковых литералов в расширенной кодировке. В остальном же поддерживаются только функции преобразования между расширенной и обычной кодировками. Этот подход также был реализован в C++ с типом символов `wchar_t`, который является вполне самостоятельным типом (в отличие от языка C). По уровню библиотечной поддержки C++ превосходит C – практически все возможности типа `char` доступны для `wchar_t` (и любого другого типа, который может использоваться в качестве типа символов).

Использование локальных контекстов

Полноценная интернационализация обычно не ограничивается преобразованием текстовых сообщений. Например, необходимо также позаботиться об использовании разных стандартов форматирования чисел, денежных величин и дат. Если

некоторая функция работает с буквами, она должна на основании данных локального контекста обеспечить корректную обработку всех букв данного языка.

Согласно стандартам POSIX и X/Open, локальный контекст может задаваться и в программах языка C. Для этой цели применяется функция `setlocale()`. Смена локального контекста влияет на работу функций, зависящих от классификации и преобразования символов (например, `isupper()` и `toupper()`), а также функций ввода-вывода (таких, как `printf()`).

Однако подход, реализованный в C, ограничен. Поскольку локальный контекст является глобальным свойством, одновременное использование нескольких локальных контекстов (например, чтение вещественных чисел по английским стандартам и вывод их в немецком стандарте) либо невозможно, либо требует относительно больших усилий. Кроме того, локальные контексты C не расширяются. Они обладают только теми возможностями, которые заложены в них реализацией. Если потребуется реализовать какую-нибудь новую возможность, придется использовать другой механизм. Наконец, механизм C не позволяет определять новые локальные контексты для поддержки специальных культурных стандартов.

Стандартная библиотека C++ решает все перечисленные проблемы при помощи объектно-ориентированного подхода. Прежде всего, строение локального контекста инкапсулируется в объекте типа `locale`, что позволяет использовать несколько локальных контекстов одновременно. Операции, зависящие от локальных контекстов, настраиваются на применение соответствующих объектов. Например, с каждым потоком данных для ввода-вывода можно ассоциировать объект локального контекста, который действует различными функциями потока данных для соблюдения соответствующих стандартов. В следующем примере показано, как это делается:

```
// i18n/loc1.cpp
#include <iostream>
#include <locale>
using namespace std;

int main()
{
    // Использование классического локального контекста C
    // для чтения данных из стандартного ввода
    cin.imbue(locale::classic());

    // Использование немецкого локального контекста
    // для записи данных в стандартный вывод
    cout.imbue(locale("de_DE"));

    // Чтение и запись вещественных чисел в цикле
    double value;
    while (cin >> value) {
        cout << value << endl;
    }
}
```

Показанная ниже команда ассоциирует «классический» локальный контекст С со стандартным каналом ввода:

```
cin.imbue(locale::classic());
```

В этом локальном контексте форматирование дат и чисел, классификация символов и т. д. выполняются так же, как в исходном языке С без назначения локальных контекстов. Следующее выражение возвращает соответствующий объект класса `locale`:

```
std::locale::classic()
```

Тот же результат будет получен при использовании выражения:

```
std::locale("C")
```

Это выражение конструирует объект `locale` с заданным именем. Специальное имя "С" является единственным именем, которое заранее поддерживается реализацией C++. Поддержка любых других локальных контекстов не гарантирована, хотя предполагается, что язык C++ поддерживает другие локальные контексты.

Соответственно следующая команда ассоциирует локальный контекст `de_DE` со стандартным каналом ввода:

```
cout.imbue(locale("de_DE"));
```

Разумеется, команда выполняется успешно только в том случае, если этот локальный контекст поддерживается системой. Если при конструировании объекта локального контекста будет указано неизвестное имя, генерируется исключение `runtime_error`.

Если все прошло успешно, то входные данные читаются по классическим правилам С (например, 47.11), а выходные данные выводятся по немецким стандартам (47.11). В Германии, как и в России, для отделения дробной части от целой используется запятая.

Как правило, программы не определяют свои локальные контексты, кроме случаев, когда чтение и запись данных производятся в фиксированном формате. Вместо этого локальный контекст определяется по значению переменной окружения `LANG`. Имя локального контекста также может вводиться из других источников, как показано в следующем примере:

```
// i18n/loc2.cpp
#include <iostream>
#include <locale>
#include <string>
#include <cstdlib>
using namespace std;

int main()
{
    // Создание локального контекста по умолчанию
    // в зависимости от состояния переменной окружения
    locale langLocale("");
```

```

// Присваивание локального контекста стандартному каналу вывода
cout.imbue(langLocale);

// Обработка имени локального контекста
bool isGerman;
if (langLocale.name() == "de_DE" ||
    langLocale.name() == "de" ||
    langLocale.name() == "german") {
    isGerman = true;
}
else {
    isGerman = false;
}

// Чтение локального контекста для ввода
if (isGerman) {
    cout << "Sprachumgebung fuer Eingaben: ";
}
else {
    cout << "Locale for input: ";
}
string s;
cin >> s;
if (!cin) {
    if (isGerman) {
        cerr << "FEHLER beim Einlesen der Sprachumgebung"
            << endl;
    }
    else {
        cerr << "ERRDR while reading the locale" << endl;
    }
    return EXIT_FAILURE;
}
locale cinLocale(s.c_str());

// Присваивание локального контекста стандартному каналу ввода
cin.imbue(cinLocale);

// Чтение и запись вещественных чисел в цикле
double value;
while (cin >> value) {
    cout << value << endl;
}

```

В этом примере следующая команда создает объект класса `locale`:

```
locale langLocale("");
```

Пустая строка вместо имени локального контекста имеет особый смысл: она обозначает локальный контекст по умолчанию для окружения пользователя (обычно определяется переменной окружения `LANG`). Этот локальный контекст связывается со стандартным входным потоком данных командой:

```
cout.imbue(langLocale);
```

Показанное ниже выражение возвращает имя локального контекста по умолчанию в виде объекта типа `string` (см. главу 11):

```
langLocale.name()
```

Следующий фрагмент конструирует локальный контекст по имени, прочитанному из стандартного входного потока данных:

```
string s;
cin >> s;
...
locale cinLocale(s.c_str());
```

Для этого из стандартного входного потока данных читается слово, передаваемое в аргументе конструктора. Если попытка чтения завершается неудачей, во входном потоке данных устанавливается флаг `ios_base::failbit`. Программа проверяет и обрабатывает эту ситуацию:

```
if (!cin) {
    if (isGerman) {
        cerr << "FEHLER beim Einlesen der Sprachumgebung"
            << endl;
    }
    else {
        cerr << "ERROR while reading the locale" << endl;
    }
    return EXIT_FAILURE;
}
```

Если содержимое строки не позволяет сконструировать локальный контекст, также генерируется исключение `runtime_error`.

Если программист хочет соблюдать национальные стандарты, он должен задействовать соответствующие объекты локальных контекстов. Статическая функция `global()` класса `locale` устанавливает новый глобальный объект локального контекста. Этот объект будет использоваться по умолчанию функциями, которым при вызове может передаваться необязательный аргумент с локальным контекстом. Если объект локального контекста, назначенный функцией `global()`, обладает именем, вызов `global()` также обеспечивает правильную реакцию со стороны функций C с поддержкой локальных контекстов. Если имя отсутствует, то результаты вызова функций C зависят от реализации.

Пример назначения глобального объекта локального контекста в зависимости от окружения, в котором выполняется программа:

```
std::locale::global(std::locale(""));
```

В частности, информация о назначенному контексте передается функциям С, которые будут вызываться в будущем. Последствия будут такими же, как при следующем вызове:

```
std::setlocale(LC_ALL, "")
```

Тем не менее глобальное назначение локального контекста не заменяет локальные контексты, уже хранящиеся в объектах. Оно лишь заменяет объект локального контекста, копируемый при создании контекста конструктором по умолчанию. Например, вызов `locale::global()` не влияет на объекты локальных контекстов, хранящиеся в объектах потоков данных. Если вы хотите, чтобы существующий поток данных использовал конкретный локальный контекст, свяжите его с этим контектом при помощи функции `imbue()`.

Глобально назначенный контекст используется при создании объектов локального контекста конструктором по умолчанию. В этом случае новый контекст представляет собой копию глобально назначенного контекста на момент конструирования. Следующий фрагмент назначает контекст по умолчанию для стандартных потоков данных:

```
// Глобальная регистрация объекта локального контекста для потоков
std::cin.imbue(std::locale());
std::cout.imbue(std::locale());
std::cerr.imbue(std::locale());
```

При использовании локальных контекстов в С++ важно помнить, что они почти не связаны с локальными контекстами С. Твердо уверенным можно быть только в одном: единый локальный контекст С изменяется при глобальном назначении именованного объекта локального контекста С++. В общем случае нельзя предполагать, что функции С и С++ работают в одних и тех же локальных контекстах.

Фацеты

На функциональном уровне локальный контекст делится на несколько специальных объектов. Объект, обеспечивающий работу некоторого аспекта интернационализации, называется *фацетом*. Это означает, что объект локального контекста может рассматриваться как контейнер для различных фацетов. Для обращения к некоторому аспекту локального контекста тип соответствующего фацета передается в аргументе шаблонной функции `use_facet()`. Например, следующее выражение обращается к фацету типа `numruncst` для типа символов `char` объекта локального контекста `loc`:

```
std::use_facet<std::numruncst<char>>(loc)
```

Каждый тип фацета определяется в виде класса, предоставляющего определенные сервисы. Например, тип фацета `numruncst` предоставляет сервис форматирования числовых и логических величин. Так, следующее выражение возвращает строку, используемую для представления `true` в локальном контексте `loc`:

```
std::use_facet<std::numruncst<char>>(loc).truename()
```

В табл. 14.3 приведена сводка фасетов, поддерживаемых стандартной библиотекой C++. Фасеты делятся на категории, используемые некоторыми конструкторами для создания новых объектов локальных контекстов на основании комбинации других объектов.

Таблица 14.3. Типы фасетов, определенные в стандартной библиотеке C++

Категория	Тип фасета	Использование
numeric	num_get<>()	Ввод числовых данных
	num_put<>()	Вывод числовых данных
	numruneget<>()	Символы, используемые при числовом вводе-выводе
time	time_get<>()	Ввод даты и времени
	time_put<>()	Вывод даты и времени
monetary	money_get<>()	Ввод денежных величин
	money_put<>()	Вывод денежных величин
	moneyruneget<>()	Символы, используемые при вводе-выводе денежных величин
ctype	ctype<>()	Информация о символах (toupper(), isupper())
	codecvt<>()	Преобразование между кодировками
collate	collate<>()	Контекстное сравнение строк
messages	messages<>()	Чтение строковых сообщений

Определение собственных версий фасетов позволяет создавать специализированные локальные контексты. В следующем примере показано, как это делается. Сначала в нем определяется фасет для использования немецких строковых представлений логических величин:

```
class germanBoolNames : public std::numrunebyname<char> {
public:
    germanBoolNames (const char *name)
        : std::numrunebyname<char>(name) {
    }
protected:
    virtual std::string do_truename () const {
        return "wahr";
    }
    virtual std::string do_falsename () const {
        return "falsch";
    }
};
```

Класс `germanBoolNames` объявлен производным от класса `numrunebyname`, определенного в стандартной библиотеке C++. Этот класс определяет параметры локального контекста, применяемые при числовом форматировании. Использование базового класса `numrunebyname` вместо `numrune` позволяет модифицировать функции класса, которые не переопределяются явным образом. Значения,

возвращаемые этими функциями, зависят от имени, переданного в аргументе конструктора. Если бы в качестве базового использовался класс `numprint`, то поведение этих функций было бы жестко фиксированным. Класс `germanBoolNames` переопределяет две функции, определяющие текстовое представление значений `true` и `false`.

Чтобы использовать этот фасет в локальном контексте, необходимо создать новый объект локального контекста при помощи специального конструктора класса `locale`. В первом аргументе этого конструктора передается объект локального контекста, а во втором — указатель на фасет. Созданный локальный контекст идентичен первому аргументу во всем, кроме фасета, переданного во втором аргументе. Заданный фасет устанавливается в созданном объекте контекста после копирования первого аргумента:

```
std::locale loc (std::locale(""). new germanBoolNames());
```

Подвыражение `new` создает фасет, устанавливаемый в новом локальном контексте. Таким образом, фасет регистрируется в `loc` для создания специализированной версии `locale("")`. Поскольку прямая модификация объектов локального контекста невозможна, для установки нового фасета в локальном контексте придется создать новый объект локального контекста. Этот новый объект используется так же, как любой другой объект локального контекста. Например, рассмотрим такой фрагмент:

```
std::cout.imbue(loc);
std::cout << std::boolalpha << true << std::endl;
```

Этот фрагмент выдает следующий результат:

```
wahr
```

Вы также можете создать абсолютно новый фасет. В этом случае функция `has_facet()` позволяет узнать, был ли этот новый фасет зарегистрирован для заданного объекта локального контекста.

Строение объекта локального контекста

Локальный контекст C++ представляет собой неизменяемый контейнер для фасетов. Класс `locale` определяется в заголовочном файле `<locale>` следующим образом:

```
namespace std {
    class locale {
        public:
            // Глобальные объекты локальных контекстов
            static const locale& classic(); // Классический локальный контекст С
            static      locale global(const locale&); // Глобальное назначение
                                                // локального контекста
            // Внутренние типы и значения
            class facet;
            class id;
    };
}
```

```
typedef int category;
static const category none, numeric, time, monetary,
    ctype, collate, messages, all;

// Конструкторы
locale() throw();
explicit locale (const char* name);

// Создание локального контекста на базе существующих контекстов
locale (const locale& loc) throw();
locale (const locale& loc, const char* name, category);
template <class Facet>
    locale (const locale& loc, Facet* fp);
locale (const locale& loc, const locale& loc2, category);

// Оператор присваивания
const locale& operator= (const locale& loc) throw();
template <class Facet>
    locale combine (const locale& loc);

// Деструктор
~locale() throw();

// Имя (если есть)
basic_string<char> name() const;

// Сравнения
bool operator== (const locale& loc) const;
bool operator!= (const locale& loc) const;

// Сортировка строк
template <class charT, class Traits, class Allocator>
    bool operator() (
        const basic_string<charT,Traits,Allocator>& s1,
        const basic_string<charT,Traits,Allocator>& s2) const;
};

// Работа с фацетами
template <class Facet>
    const Facet& use_facet (const locale&);

template <class Facet>
    bool has_facet (const locale&) throw();
}
```

У локальных контекстов есть одна специфическая особенность — механизм обращения к объектам фацетов, хранящимся в контейнере. Обращение к фацету локального контекста производится по типу этого фацета, который интерпретируется как индекс. Поскольку фацеты обладают разными интерфейсами и предназначаются для разных целей, желательно, чтобы функции доступа к локальным

контекстам возвращали тип, соответствующий индексу; именно это и происходит при «индексировании» по типу фацета. Другое достоинство этого механизма состоит в том, что интерфейс безопасен по отношению к типам.

Объекты локальных контекстов неизменны. Это означает, что фацеты, хранящиеся в локальных контекстах, невозможно модифицировать (кроме присваивания локальных контекстов). Модификации локальных контекстов создаются посредством объединения существующих контекстов и фацетов с формированием нового контекста. В табл. 14.4 перечислены конструкторы локальных контекстов.

Таблица 14.4. Конструкторы локальных контекстов

Выражение	Описание
<code>locale()</code>	Создает копию текущего локального контекста, назначенного глобально
<code>locale(name)</code>	Создает объект локального контекста для заданного имени
<code>locale(loc)</code>	Создает копию локального контекста <code>loc</code>
<code>locale(loc, loc2, cat)</code>	Создает копию локального контекста <code>loc1</code> , в которой все фацеты из категории <code>cat</code> заменяются фацетами локального контекста <code>loc2</code>
<code>locale(loc, name, cat)</code>	Эквивалент <code>locale(loc, locale(name), cat)</code>
<code>locale(loc,fp)</code>	Создает копию локального контекста <code>loc</code> и устанавливает фацет, на который ссылается указатель <code>fp</code>
<code>loc1 = loc2</code>	Присваивает локальный контекст <code>loc2</code> объекту <code>loc1</code>
<code>loc1.template combine<F>(loc2)</code>	Создает копию локального контекста <code>loc1</code> , в которой фацет типа <code>F</code> берется из <code>loc2</code>

Почти все конструкторы создают копию другого объекта локального контекста. Простое копирование объекта считается относительно дешевой операцией. Фактически оно сводится к заданию указателя и увеличению счетчика ссылок. Создание модифицированного локального контекста обходится дороже, поскольку при этом приходится изменять счетчики ссылок для всех фацетов, хранящихся в локальном контексте. Хотя стандарт не дает гарантий относительно эффективности этой операции, весьма вероятно, что во всех реализациях копирование локальных контекстов будет производиться достаточно эффективно.

Два конструктора из табл. 14.4 получают имена локальных контекстов. Передаваемые имена не стандартизированы (кроме имени `C`). Однако стандарт требует, чтобы в документации, прилагаемой к стандартной библиотеке C++, были перечислены допустимые имена. Предполагается, что большинство реализаций поддерживает имена, представленные на с. 664.

Функцию `combine()` стоит пояснить подробнее, поскольку в ней используется возможность, относительно недавно реализованная в компиляторах. Она представляет собой шаблонную функцию с явно заданным аргументом. Это означает, что тип аргумента шаблона не определяется по значению аргумента, поскольку такого значения попросту нет. Вместо этого аргумент шаблона (в данном случае — тип `F`) задается явно.

В двух функциях для обращения к фацетам объекта локального контекста используется одинаковая методика (табл. 14.5). Однако эти две функции являются глобальными шаблонными функциями, поэтому уродливый синтаксис с ключевым словом `template` становится излишним.

Таблица 14.5. Работа с фацетами

Выражение	Описание
<code>has_facet<F>(loc)</code>	Возвращает <code>true</code> , если локальный контекст <code>loc</code> содержит фацет типа <code>F</code>
<code>use_facet<F>(loc)</code>	Возвращает ссылку на фацет типа <code>F</code> , хранящийся в локальном контексте <code>loc</code>

Функция `use_facet()` возвращает ссылку на фацет. Тип ссылки соответствует типу, переданному явно в аргументе шаблона. Если локальный контекст, переданный в аргументе, не содержит соответствующего фацета, функция генерирует исключение `bad_cast`. Чтобы узнать, присутствует ли в заданном локальном контексте некоторый фацет, воспользуйтесь функцией `has_facet()`.

Остальные операции локальных контекстов перечислены в табл. 14.6. Имя локального контекста поддерживается в том случае, если контекст был сконструирован по заданному имени или одному (или нескольким) именованным локальным контекстам. Однако и в этом случае стандарт не дает гарантий относительно построения 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 одинаковые имена. Было бы естественно считать, что два объекта идентичны, если один из них является копией другого. Но что делать с именами? Предполагается, что имя локального контекста отражает имена, используемые для конструирования именованных фацетов. Например, имя локального контекста может быть сконструировано объединением имен фацетов в определенном порядке, разделенных заданными символами. Вероятно, такая схема позволит определить идентичность двух объектов локального контекста, если они были сконструированы посредством объединения одних и тех же именованных фацетов. Иначе говоря, стандарт требует, чтобы два локальных контекста, содержащих одинаковые именованные фацеты, считались идентичными.

Таблица 14.6. Операции с локальными контекстами

Выражение	Описание
<code>loc.name()</code>	Возвращает имя локального контекста <code>loc</code> в виде <code>string</code>
<code>loc1 == loc2</code>	Возвращает <code>true</code> , если локальные контексты <code>loc1</code> и <code>loc2</code> идентичны
<code>loc1 != loc2</code>	Возвращает <code>true</code> , если локальные контексты <code>loc1</code> и <code>loc2</code> не идентичны
<code>loc(str1,str2)</code>	Возвращает логический признак сравнения строк <code>str1</code> и <code>str2</code> в порядковом отношении (то есть проверяет, что <code>str1</code> меньше <code>str2</code>)
<code>locale::classic()</code>	Возвращает <code>locale("C")</code>
<code>locale::global(loc)</code>	Назначает локальный контекст <code>loc</code> как глобальный и возвращает предыдущий глобально назначенный локальный контекст

Оператор () позволяет использовать объект локального контекста для сравнения строк. При этом строки, переданные в аргументах, сравниваются в порядковом отношении, и действует фасет `collate`. Иначе говоря, фасет `collate` проверяет, будет ли одна строка меньше другой по критериям объекта локального контекста. Такое поведение типично для объектов функций STL (см. с. 295), поэтому объект локального контекста может использоваться в качестве критерия сортировки для алгоритмов STL, работающих со строками. Например, сортировка вектора по правилам немецкого локального контекста выполняется следующим образом:

```
std::vector<std::string> v;  
// Сортировка строк в немецком локальном контексте  
std::sort (v.begin(), v.end(), // Интервал  
           locale("de_DE")); // Критерий сортировки
```

Строение фасетов

Возможности локального контекста определяются содержащимися в нем фасетами. Все локальные контексты заведомо содержат минимальный набор стандартных фасетов. В описаниях отдельных фасетов, приведенных далее, указано, какие специализации гарантированно присутствуют в контексте. Помимо перечисленных реализация стандартной библиотеки C++ может включить в локальный контекст дополнительные фасеты. Важно понимать, что пользователь также может добавить собственные фасеты или установить их вместо стандартных фасетов.

На с. 670 рассматривается процедура установки фасета в локальном контексте. Например, класс `germanBoolNames` был объявлен производным от класса `numpunct_byname<char>` одного из стандартных фасетов и установлен в локальном контексте при помощи конструктора, в аргументах которого передаются локальный контекст и фасет. Но что нужно для того, чтобы создать собственный фасет? В качестве фасета может использоваться любой класс F, удовлетворяющий двум требованиям.

- Класс F является открытым производным от класса `locale::facet`. Базовый класс в основном определяет механизмы подсчета ссылок, используемые во внутренней работе объектов локального контекста. Кроме того, он объявляет закрытыми копирующий конструктор и оператор присваивания, что предотвращает копирование или присваивание фасетов.
- Класс F содержит открытую статическую переменную `id` типа `locale::id`. Эта переменная используется для поиска фасета в объекте контекста по его типу. Применение типа в качестве индекса прежде всего направлено на обеспечение типовой безопасности интерфейса. Во внутреннем представлении для работы с фасетами используется обычный контейнер с целочисленными индексами.

Стандартные фасеты соответствуют не только этим требованиям, но и некоторым специальным рекомендациям — не обязательным, но весьма полезным.

- Все функции класса объявляются константными. Это полезно, поскольку функция `use_facet()` возвращает ссылку на константный фасет. Функции, не объявленные константными, вызываться не могут.
- Все открытые функции объявляются невиртуальными и передают запросы защищенным виртуальным функциям. Имя защищенной функции совпадает с именем открытой функции, с добавлением префикса `do_`. Например, функция `numrunc::truename()` вызывает функцию `numrunc::do_truename()`. Подобная схема выбора имен помогает избежать замещения функций при переопределении только одной из нескольких виртуальных функций с одинаковыми именами. Например, класс `num_put` содержит несколько функций с именем `put`. Кроме того, программист базового класса может включить в невиртуальные функции дополнительный код, который будет выполняться даже в случае переопределения виртуальных функций.

Следующее описание стандартных фасетов относится только к открытым функциям. Чтобы изменить фасет, всегда приходится переопределять соответствующие защищенные функции. Определение функций с таким же интерфейсом, как у открытых функций фасета, всего лишь перегрузит их, поскольку эти функции не являются виртуальными.

Для большинства стандартных фасетов определяется версия с суффиксом `_byname`. Она является производной от стандартного фасета и создает специализацию для соответствующего имени локального контекста. Так, класс `numrunc_byname` создает фасет `numrunc` для локального контекста с заданным именем. Например, команда создания немецкого фасета `numrunc` может выглядеть так:

```
std::numrunc_byname("de_DE")
```

Классы `_byname` создаются в процессе внутренней работы конструкторов локального контекста, получающих имя в виде аргумента. Для каждого стандартного фасета, поддерживающего имя, конструирование экземпляра этого фасета производится соответствующим классом `_byname`.

Числовое форматирование

Средства числового форматирования преобразуют числа из внутреннего представления в соответствующее текстовое представление. Операторы потоков данных поручают работу по непосредственному преобразованию фасетам категории `locale::numeric`. Эта категория состоит из трех фасетов:

- `numrunc` — правила оформления, используемые при форматировании и лексическом разборе чисел;
- `num_put` — форматирование чисел при выводе;
- `num_get` — лексический разбор чисел при вводе.

В двух словах, фасет `num_put` выполняет операции числового форматирования, описанные для потоков данных на с. 589, а `num_get` — лексический разбор соответствующих строк. Фасет `numrunc` предоставляет дополнительные возможности, недоступные непосредственно через интерфейс потоков данных.

Оформление

Фацет `numrunc` определяет символ, используемый в качестве десятичной точки, управляет вставкой необязательных разделителей групп разрядов, а также задает текстовые представления логических значений. Функции фацета `numrunc` перечислены в табл. 14.7.

Таблица 14.7. Функции фацета `numrunc`

Выражение	Описание
<code>pr.decimal_point()</code>	Возвращает символ, используемый в качестве десятичной точки
<code>pr.thousands_sep()</code>	Возвращает символ разделения групп разрядов
<code>pr.grouping()</code>	Возвращает объект <code>string</code> с описанием позиций разделителей
<code>pr.truename()</code>	Возвращает текстовое представление значения <code>true</code>
<code>pr.falsename()</code>	Возвращает текстовое представление значения <code>false</code>

В аргументе шаблона `numrunc` передается символьный тип `charT`. Символы, возвращаемые функциями `decimal_point()` и `thousand_sep()`, относятся к этому типу, а функции `truename()` и `falsename()` возвращают `basic_string<charT>`. Обязательна поддержка двух специализаций `numrunc<char>` и `numrunc<wchar_t>`.

Поскольку числа из множества цифр плохо читаются без промежуточных разделителей, стандартные фацеты числового форматирования и лексического разбора позволяют разделять группы разрядов. Чаще всего цифры, представляющие целое число, группируются по триплетам. Например, в США один миллион записывается так:

1.000.000

К сожалению, такое представление используется не везде. Например, в Германии для разделения группы разрядов используется точка, поэтому немец записывает это же число в несколько ином виде:

1.000.000

Символ-разделитель определяется функцией `thousands_sep()`. Но и этого недостаточно, поскольку в некоторых странах встречаются другие правила разделения разрядов. Например, в Непале один миллион записывается в виде:

10.00.000

В этом случае разные группы содержат разное количество цифр. В таких случаях может пригодиться строка, возвращаемая функцией `grouping()`. Цифра в позиции с индексом i определяет количество разрядов в i -й группе, отсчет ведется от нуля с крайней правой группы. Если количество символов в строке меньше количества групп, размер последней заданной группы применяется повторно. Значение `numeric_limits<char>::max()` определяет группу неограниченного размера, а при полном отсутствии групп используется пустая строка. В табл. 14.8 приведены примеры разного форматирования одного числа.

Таблица 14.8. Примеры разделения разрядов в одном миллионе

Строка	Результат
{ 0 } или "" (используется по умолчанию)	1000000
{ 3, 0 } или {3}	1,000,000
{ 3, 2, 3, 0 } или "\3\2\3"	10,00,000
{ 2, CHAR_MAX, 0 }	10000,00

Обратите внимание, что обычные цифры в данном случае практически бесполезны. Например, строка "2" для кодировки ASCII определяет группы в 50 цифр, потому что символу 2 в кодировке ASCII соответствует целочисленный код 50.

Форматирование

Фасет `num_put` обеспечивает текстовое форматирование чисел. Он представляет собой шаблон класса с двумя аргументами: тип `charT` определяет символы, создаваемые при выводе, а тип `OutIt` определяет итератор вывода для записи генерированных символов. По умолчанию итератор вывода относится к типу `ostreambuf_iterator<charT>`. Фасет `num_put` поддерживает семейство функций, которые называются `put()` и различаются только по последнему аргументу. Пример использования фасета `num_put`:

```
std::locale loc;
DutIt to = ...;
std::ios_base& fmt = ...;
charT fill = ...;
T value = ...;

// Получение фасета числового форматирования для контекста loc
const std::num_put<charT,DutIt>& np
= std::use_facet<std::num_put<charT,OutIt>>(loc);

// Вывод данных с использованием фасета
np.put(to, fmt, fill, value);
```

Приведенный фрагмент строит текстовое представление значения `value` с помощью символов типа `charT` и выводит их через итератор вывода `to`. Формат вывода определяется флагами форматирования, хранящимися в `fmt`, а символ `fill` используется в качестве заполнителя. Функция `put()` возвращает итератор для позиции, следующей за последним из выведенных символов.

Фасет `num_put` содержит функции для типов `bool`, `long`, `unsigned long`, `double`, `long double` и `void*`, передаваемых в последнем аргументе, и не содержит функций для таких типов, как `short` или `int`, но это не вызывает проблем, поскольку соответствующие значения базовых типов в случае необходимости автоматически преобразуются к поддерживаемым типам.

Стандарт требует, чтобы в каждом локальном контексте хранились две специализации `num_put<char>` и `num_put<wchar_t>` (в обеих специализациях для второго аргумента используется значение по умолчанию). Кроме того, стандарт-

ная библиотека C++ поддерживает все специализации, у которых в первом аргументе шаблона передается тип символа, а во втором — тип итератора вывода. Конечно, стандарт не требует, чтобы эти специализации хранились в каждом локальном контексте, потому что в этом случае количество фацетов оказалось бы бесконечным.

Лексический разбор

Фацет `num_get` предназначен для лексического разбора текстовых представлений чисел. По аналогии с `num_put` он оформляется в виде шаблона с двумя аргументами: типом символов `charT` и типом итератора ввода `InIt`, который по умолчанию равен `istreambuf_iterator<charT>`. Фацет поддерживает набор функций `get()`, различающихся только по типу последнего аргумента. Пример использования фацета `num_get`:

```
std::locale loc; // Локальный контекст
InIt beg = ...; // Начало входной последовательности
InIt end = ...; // Конец входной последовательности
std::ios_base& fmt = ...; // Поток, определяющий формат ввода
std::ios_base::iostate err; // Состояние после вызова
T value = ...; // Значение после успешного вызова

// Получение фацета разбора числовых данных для контекста loc
const std::num_get<charT, InIt>& ng
= std::use_facet<std::num_get<charT, InIt>>(loc);

// Чтение данных с использованием фацета
ng.get(beg, end, fmt, err, value);
```

В этом фрагменте делается попытка выполнить разбор числового значения, соответствующего типу `T`, из последовательности символов между `beg` и `end`. Формат предполагаемого числового значения определяется аргументом `fmt`. Если разбор завершается неудачей, в переменной `err` устанавливается флаг `ios_base::failbit`; если все прошло нормально, состояние `ios_base::goodbit` сохраняется в `err`, а полученное значение — в `value`. Переменная `value` изменяется только в случае успешного разбора. Если вся последовательность символов была полностью использована, функция `get()` возвращает второй параметр (`end`). В противном случае возвращается итератор, указывающий на первый символ, который не был обработан как часть числового значения.

Фацет `num_get` содержит функции чтения для типов `bool`, `long`, `unsigned short`, `unsigned int`, `unsigned long`, `float`, `double` и `long double`. Некоторые из этих типов (например, `unsigned short`) не имеют соответствующих функций в фацете `num_put`. Дело в том, что вывод значения типа `unsigned short` дает такой же результат, что и вывод значения типа `unsigned short`, приведенного к типу `unsigned long`. С другой стороны, чтение значения типа `unsigned long` с последующим преобразованием к `unsigned short` может дать результат, отличный от прямого чтения `unsigned short`.

Стандарт требует, чтобы в каждом локальном контексте хранились две специализации `num_get<char>` и `num_get<wchar_t>` (в обеих специализациях для второго аргумента используется значение по умолчанию). Кроме того, стандарт-

ная библиотека C++ поддерживает все специализации, у которых в первом аргументе шаблона передается тип символа, а во втором — тип итератора вывода. Как и в случае с `num_put`, стандарт не требует, чтобы эти специализации хранились в каждом локальном контексте.

Форматирование времени и даты

Фацеты `time_get` и `time_put` из категории `time` предоставляют средства разбора и форматирования времени и даты. Функции, выполняющие эти операции, работают с объектами типа `tm`, определяемого в заголовочном файле `<ctime>`. Объекты не передаются напрямую; в аргументах передаются указатели на них.

Оба фацета из категории `time` в значительной степени зависят от функции `strftime()` (также определяемой в заголовочном файле `<ctime>`). Преобразование объекта `tm` к текстовому представлению описывается строкой, содержащей спецификаторы формата. Краткая сводка спецификаторов приведена в табл. 14.9. Эти же спецификаторы используются фацетом `time_put`.

Таблица 14.9. Спецификаторы формата для функции `strftime()`

Спецификатор	Смысл	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 для текущего локального контекста	Jul 12 21:53:22 1998
%d	День месяца	12
%H	Час (по 24-часовой шкале)	21
%I	Час (по 12-часовой шкале)	9
%j	День года	193
%m	Порядковый номер месяца	7
%M	Минуты	53
%p	Обозначение половины суток (am или pm)	pm
%S	Секунды	22
%U	Номер недели, начиная с первого воскресенья	28
%W	Номер недели, начиная с первого понедельника	28
%w	Порядковый номер дня недели (воскресенье=0)	0
%x	Представление даты для текущего локального контекста	Jul 12 1998
%X	Представление времени для текущего локального контекста	21:53:22

продолжение ↴

Таблица 14.9 (продолжение)

Спецификатор	Смысл	Пример
%y	Год (без указания века)	98
%Y	Год (с указанием века)	1998
%Z	Часовой пояс	MEST
%%	Литерал %	%

Конечно, точный вид строки, созданной функцией `strftime()`, зависит от текущего локального контекста. Примеры в таблице приведены для локального контекста "C".

Лексический разбор

Фацет `time_get` представляет собой шаблон, в аргументах которого передаются тип символов `charT` и тип итератора ввода `InIt`. По умолчанию итератор ввода относится к типу `istreambuf_iterator<charT>`. В табл. 14.10 перечислены функции, определенные для фацета `time_get`. Все эти функции, за исключением `date_order()`, обрабатывают строку и сохраняют результаты в объекте `tm`, на который ссылается аргумент `t`. Если попытка разбора завершается неудачей, функции либо выставляют признак ошибки (например, модификацией аргумента `err`), либо сохраняют неопределенное значение. Это означает, что надежная обработка гарантирована для времени, сгенерированного программой, но не для времени, введенного пользователем.

Таблица 14.10. Функции фацета `time_get`

Выражение	Описание
<code>tg.get_time(beg, end, fmt, err, t)</code>	Разбирает строку между <code>beg</code> и <code>end</code> как время, сгенерированное функцией <code>strftime()</code> со спецификатором <code>X</code>
<code>tg.get_date(beg, end, fmt, err, t)</code>	Разбирает строку между <code>beg</code> и <code>end</code> как время, сгенерированное функцией <code>strftime()</code> со спецификатором <code>x</code>
<code>tg.get_weekday(beg, end, fmt, err, t)</code>	Разбирает строку между <code>beg</code> и <code>end</code> как название дня недели
<code>tg.get_monthname(beg, end, fmt, err, t)</code>	Разбирает строку между <code>beg</code> и <code>end</code> как название месяца
<code>tg.get_year(beg, end, fmt, err, t)</code>	Разбирает строку между <code>beg</code> и <code>end</code> как год
<code>tg.date_order()</code>	Возвращает порядок следования компонентов даты, используемый фацетом

Все функции возвращают итератор, установленный в позицию за последним прочитанным символом. Разбор прекращается при успешном завершении или в случае ошибки (например, если строка не интерпретируется как дата).

Функция, читающая название дня недели или месяца, читает как сокращенные, так и полные названия. Если за сокращением 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.

В стандарте не сказано, должны ли при разборе года поддерживаться обозначения из двух цифр. Также не определено, какой именно год должен сопоставляться такому сокращению.

Функция `date_order()` возвращает порядок следования дня, месяца и года в строке даты. Это необходимо для некоторых дат, которые не позволяют определить нужный порядок по текстовому представлению даты. Например, 1 февраля 2003 года может выводиться как в виде 3/2/1, так и в виде 1/2/3. В классе `time_base`, базовом для фацета `time_get`, определен перечисляемый тип `dateorder` с возможными порядками следования компонентов даты. Значения этого типа перечислены в табл. 14.11.

Таблица 14.11. Значения перечисляемого типа `dateorder`

Значение	Описание
<code>no_order</code>	Определенный порядок отсутствует (например, дата по юлианскому календарю)
<code>dmy</code>	День, месяц, год
<code>mdy</code>	Месяц, день, год
<code>ymd</code>	Год, месяц, день
<code>ydm</code>	Год, день, месяц

Стандарт требует, чтобы в каждом локальном контексте хранились две специализации `time_get<char>` и `time_get<wchar_t>` (в обеих специализациях для второго аргумента используется значение по умолчанию). Кроме того, стандартная библиотека C++ поддерживает все специализации, у которых в первом аргументе шаблона передается тип символа, а во втором — тип итератора ввода. Стандарт не требует, чтобы эти специализации хранились в каждом локальном контексте.

Форматирование

Фацет `time_put` используется для форматирования времени и даты. Он представляет собой шаблон, в аргументах которого передаются тип символов `charT` и тип итератора вывода `OutIt`. По умолчанию итератор вывода относится к типу `ostreambuf_iterator` (см. с. 638).

В фацете `time_put` определены две функции `put()`, которые преобразуют дату, хранящуюся в объекте типа `tm`, в последовательность символов, записываемую через итератор вывода. Функции фацета `time_put` перечислены в табл. 14.12.

Таблица 14.12. Функции фацета `time_put`

Выражение	Описание
<code>tp.put(to, fmt, fill, t, cbegin, cend)</code>	Преобразует дату в соответствии со строкой [cbegin,cend)
<code>tp.put(to, fmt, fill, t, cvt, mod)</code>	Преобразует дату с использованием спецификатора cvt

Обе функции записывают результаты через итератор вывода *to* и возвращают итератор, установленный в позицию за последним выведенным символом. Аргумент *fmt* типа *ios_base* используется для работы с другими фасетами и передачи возможной дополнительной информации по форматированию. Символ *fill* используется для заполнения позиций. Аргумент *t* указывает на объект типа *tm*, содержащий форматируемую дату.

Вторая версия *put()* форматирует дату из объекта *tm*, на который ссылается указатель *t*, интерпретируя аргумент *cut* как спецификатор формата функции *strftime()*. Эта функция *put()* выполняет только одно преобразование — именно то, которое задано символом *cut*. Функция *put()* вызывается другой функцией *put()* для каждого обнаруженного спецификатора. Например, при вызове со спецификатором 'X' время, хранящееся в **t*, выводится через итератор вывода. Смысл аргумента *mod* не задается стандартом. Предполагается, что этот аргумент должен влиять на выполняемое преобразование, как это делается в некоторых реализациях функции *strftime()*.

Версия *put()*, в которой для управления преобразованием используется строка в интервале [*cbeg,cend*), имеет много общего с *strftime()*. Она также сканирует строку и записывает все символы, не входящие в спецификацию преобразования, через итератор вывода *to*. Встречая спецификатор формата с префиксом %, функция извлекает спецификатор с необязательным модификатором, а затем вызывает другую версию *put()* и передает спецификатор с модификатором в двух последних аргументах. После возврата управления *put()* продолжает сканировать строку.

Фасет *time_put* отличается от других тем, что в нем имеется невиртуальная функция *put()*, которая использует строку в качестве спецификатора преобразования. Эта функция не может переопределяться в классах, производных от *time_put*. Переопределение разрешено только для других функций *put()*.

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 в первом аргументе шаблона передается тип символов (*char* или *wchar_t*), а во втором — тип итератора вывода. Не существует гарантированной поддержки для специализаций, у которых в первом аргументе шаблона передаются типы, отличные от *char* и *wchar_t*. Кроме того, не гарантировано, что по умолчанию в объектах локальных контекстов будут храниться какие-либо специализации помимо *time_put<char>* и *time_put<wchar_t>*.

Форматирование денежных величин

В категорию *monetary* входят фасеты *moneypunct*, *money_get* и *money_put*. Фасет *moneypunct* определяет формат денежных величин, а два других фасета используют эту информацию для их форматирования или лексического разбора.

Оформление

Вид выводимых денежных величин определяется локальным контекстом. Форматы, используемые в разных культурных сообществах, очень отличаются друг от друга. В частности, различаются обозначения денежных единиц (которые

могут вообще отсутствовать), способы записи отрицательных и положительных сумм, режимы использования национального или международного обозначения денежной единицы, разделители групп разрядов и т. д. Чтобы обеспечить необходимую гибкость, атрибуты формата были выделены в фацет `moneyuprint`.

Фацет `moneyuprint` представляет собой шаблон с двумя аргументами: типом `char[4]` и логическим признаком, по умолчанию равным `false`. Логический признак показывает, должно ли использоваться национальное (`false`) или международное (`true`) обозначение денежной единицы. Функции фацета `moneyuprint` перечислены в табл. 14.13.

Таблица 14.13. Функции фацета `moneyuprint`

Выражение	Описание
<code>mp.decimal_point()</code>	Возвращает символ, используемый в качестве десятичной точки
<code>mp.thousand_sep()</code>	Возвращает символ, используемый для разделения групп разрядов
<code>mp.grouping()</code>	Возвращает строку, определяющую размеры групп разрядов
<code>mp.curr_symbol()</code>	Возвращает строку с обозначением денежной единицы
<code>mp.positive_sign()</code>	Возвращает строку с обозначением знака положительного числа
<code>mp.negative_sign()</code>	Возвращает строку с обозначением знака отрицательного числа
<code>mp.frac_digits()</code>	Возвращает количество цифр в дробной части
<code>mp.pos_format()</code>	Возвращает формат неотрицательных значений
<code>mp.neg_format()</code>	Возвращает формат отрицательных значений

Класс `moneyuprint` объявлен производным от класса `money_base`. В этом базовом классе определен перечисляемый тип `part`, представляющий компоненты шаблона форматирования денежных величин. Кроме того, в нем определен тип `pattern` (синоним для `char[4]`). Четыре значения типа `part`, хранящихся в этом типе, определяют структуру денежной величины. В табл. 14.14 перечислены пять допустимых значений типа `part`, которые могут использоваться в значениях типа `pattern`.

Таблица 14.14. Компоненты шаблонов форматирования денежных величин

Значение	Описание
<code>none</code>	В данной позиции могут находиться необязательные пробелы
<code>space</code>	В данной позиции обязателен хотя бы один пробел
<code>sign</code>	В данной позиции может находиться знак
<code>symbol</code>	В данной позиции может находиться обозначение денежной единицы
<code>value</code>	В данной позиции выводится значение

В фацете `moneyuprint` определены две функции, возвращающие шаблоны форматирования: `neg_format()` для отрицательных значений и `pos_format()` для неотрицательных значений. Шаблон форматирования содержит обязательные компоненты

`sign`, `symbol` и `value` и один из двух компонентов `none` или `space`. Впрочем, это не означает, что в итоге действительно будет выведен знак или символ денежной единицы. Содержимое соответствующих позиций зависит от значений, возвращаемых другими функциями фацета, и от форматных флагов, передаваемых функциям форматирования.

В денежной величине заведомо присутствует только значение. Естественно, оно находится в позиции, обозначенной компонентом `value` шаблона форматирования. Значение содержит точно `frac_digits()` цифр в дробной части, а в качестве десятичной точки используется символ `decimal_point()` (если дробная часть отсутствует, десятичная точка не ставится).

При вводе денежных величин во входных данных могут присутствовать разделители групп разрядов. Если разделители присутствуют, правильность их расположения проверяется при помощи функции `grouping()`. Если функция `grouping()` возвращает пустую строку, присутствие разделителей групп недопустимо. Группы разрядов разделяются символом, возвращаемым функцией `thousands_sep()`. Группировка производится по тем же правилам, как при числовом форматировании (см. с. 678). При выводе денежных величин разделители групп разрядов всегда вставляются в соответствии со строкой, возвращаемой функцией `grouping()`. При чтении денежных величин разделители групп не обязательны, если строка группировки не является пустой. Правильность расположения разделителей проверяется после успешного завершения разбора.

Компоненты `space` и `none` управляют расположением пробелов. Компонент `space` используется в позиции, в которой должен присутствовать по крайней мере один пробел. В процессе форматирования, если в форматных флагах установлен флаг `ios_base::internal`, в позиции компонента `space` или `none` вставляются символы-заполнители. Конечно, заполнение производится только в том случае, если заданная минимальная ширина не заполнена другими символами. Символ-заполнитель передается в аргументе функций форматирования денежных величин. Если отформатированное значение не содержит пробелов, `none` может находиться в последней позиции шаблона форматирования. Компоненты `space` и `none` не могут находиться в первой позиции шаблона форматирования, а `space` не может находиться в последней позиции.

Знак денежной величины (положительный/отрицательный) может представляться несколькими символами. Например, в некоторых контекстах отрицательные суммы заключаются в круглые скобки. В позиции компонента `sign` в шаблоне форматирования выводится первый символ представления знака, остальные символы выводятся в конце после всех остальных компонентов. Если представление знака равно пустой строке, знак не отображается. Символьное представление знака определяется функцией `positive_sign()` для неотрицательных величин и функцией `negative_sign()` для отрицательных.

В позиции компонента `symbol` отображается символ денежной единицы. Оно присутствует только в том случае, если среди форматных флагов, используемых при выводе или разборе, установлен флаг `ios_base::showbase`. В качестве представления денежного знака используется строка, возвращаемая функцией `curr_symbol()`. Если второй аргумент шаблона равен `false` (по умолчанию), то задействуется национальное обозначение денежной единицы; иначе — международное обозначение.

В табл. 14.15 приведены примеры форматирования суммы \$-1234.56. Конечно, предполагается, что `frac_digits()` возвращает 2. Кроме того, во всех случаях используется нулевая ширина поля.

Таблица 14.15. Примеры использования шаблонов форматирования

Шаблон форматирования	Знак	
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>`. Стандартная библиотека C++ не поддерживает другие специализации.

Форматирование

Фасет `money_put` обеспечивает форматирование денежных величин при выводе. Он представляет собой шаблон, в аргументах которого передаются тип символов `charT` и тип итератора вывода `OutIt`. По умолчанию итератор вывода относится к типу `ostreambuf_iterator`. Две функции `put()` генерируют последовательность символов в соответствии с форматом, определяемым фасетом `moneypunct`. Форматируемое значение передается в виде типа `long double` или `basic_string<charT>`. Пример использования фасета:

```
// Получение фасета money_put для локального контекста loc
const std::money_put<charT, OutIt>& mp
= std::use_facet<std::money_put<charT, OutIt>>(loc);

// Вывод данных с использованием фасета
mp.put(to, intl, fmt, fill, value);
```

Аргумент `to` содержит итератор вывода типа `OutIt`, через который записывается отформатированная строка. Функция `put()` возвращает объект этого типа, установленный в позицию за последним сгенерированным символом. Аргумент `intl` определяет используемое обозначение денежной единицы (национальное международное). Аргумент `fmt` определяет флаги форматирования, в том числе ширину поля и фасет `moneypunct`, задающий формат выводимого значения. Символ `fill` используется в качестве заполнителя.

Аргумент `value` относится к типу `long double` или `basic_string<charT>`; в нем передается форматируемое значение. Если аргумент передается в строковом виде, строка может содержать только десятичные цифры и, возможно, минус в начале числа. Если строка начинается с минуса, то значение форматируется как отрицательное. После того как функция определяет отрицательное число, начальный минус отбрасывается. Количество цифр в дробной части определяется функцией `frac_digits()` фасета `moneypunct`.

Стандарт требует, чтобы в каждом локальном контексте хранились специализации `money_put<char>` и `money_put<wchar_t>`. Кроме того, стандартная библиотека C++ поддерживает все специализации, у которых в первом аргументе шаблона передается тип символа, а во втором — тип итератора вывода. Хранение таких специализаций в каждом локальном контексте не является обязательным.

Лексический разбор

Фасет `moneyp_get` предназначен для разбора текстовых представлений денежных величин. Он представляет собой шаблон с двумя аргументами: типом символов `charT` и типом итератора ввода `InIt`, который по умолчанию равен `istreambuf_iterator<charT>`. В классе определены две функции `get()`, которые пытаются разобрать вводимое значение и, если попытка завершается успешно, сохраняют результат в переменной типа `long double` или `basic_string<charT>`. Пример использования фасета `moneyp_get`:

```
// Получение фасета разбора денежных величин для контекста loc
const std::moneyp_get<charT, InIt>& mg
    = std::use_facet<std::moneyp_get<charT, InIt>>(loc);

// Чтение данных с использованием фасета
mg.get(beg, end, fmt, err, value);
```

Разбираемая последовательность символов находится между значениями, заданными аргументами `beg` и `end`. Разбор прекращается после обработки всех элементов или при ошибке. Если разбор завершается неудачей, в переменной `err` устанавливается флаг `ios_base::failbit`, а в переменной, задаваемой в аргументе `val`, ничего не сохраняется. В случае успешного выполнения результат сохраняется в переменной типа `long double` или `basic_string`, передаваемой по ссылке в аргументе `val`.

Аргумент `intl` содержит логический признак выбора обозначения денежной единицы (национального или международного). Фасет `moneypunct`, определяющий формат разбираемого значения, задается для объекта локального контекста, связанного с аргументом `fmt`. При разборе денежных величин всегда используется шаблон форматирования, возвращаемый функцией `neg_format()` фасета `moneypunct`.

В позиции `pop` или `space` функция, выполняющая разбор денежной величины, обрабатывает все доступные символы (кроме ситуации, когда `pop` находится в последней позиции шаблона форматирования). Завершающие пробелы не игнорируются. Функции `get()` возвращают итератор, установленный в позицию за последним обработанным символом.

Стандарт требует, чтобы в каждом локальном контексте хранились специализации `moneyp_get<char>` и `moneyp_get<wchar_t>`. Кроме того, стандартная би-

библиотека C++ поддерживает все специализации, у которых в первом аргументе шаблона передается тип `char` или `wchar_t`, а во втором — тип итератора ввода. Хранение таких специализаций в каждом локальном контексте не обязательно.

Классификация и преобразования 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, проверки того, является ли символ буквой). Кроме того, в нем предусмотрены методы смены регистра символов, а также преобразования между `char` и типом символов, для которого был специализирован данный фацет. Фацет `codecvt` обеспечивает смену кодировок символов и используется в основном шаблоном `basic_filebuf` для преобразования между внутренними и внешними представлениями.

Классификация символов

Фацет `ctype` представляет собой шаблон, параметризованный по типу символов. Класс `ctype<charT>` поддерживает три категории функций:

- функции преобразования между типами `char` и `charT`;
- функции классификации символов;
- функции преобразования к верхнему и нижнему регистрам.

Функции фацета `ctype` перечислены в табл. 14.16.

Таблица 14.16. Функции фацета `ctype`

Выражение	Описание
<code>ct.is(m, c)</code>	Проверяет, соответствует ли символ с маске <code>m</code>
<code>ct.is(beg, end, vec)</code>	Для каждого символа в интервале между <code>beg</code> и <code>end</code> сохраняет маску символа в соответствующей позиции <code>vec</code>
<code>ct.scan_is(m, beg, end)</code>	Возвращает указатель на первый символ в интервале между <code>beg</code> и <code>end</code> , соответствующий маске <code>m</code> , или <code>end</code> при отсутствии таких символов
<code>ct.scan_not(m, beg, end)</code>	Возвращает указатель на первый символ в интервале между <code>beg</code> и <code>end</code> , не соответствующий маске <code>m</code> , или <code>end</code> , если все символы соответствуют заданной маске
<code>ct.toupper(c)</code>	Возвращает букву верхнего регистра, соответствующую символу <code>c</code> . Если такой буквы не существует, возвращает <code>c</code>
<code>ct.toupper(beg, end)</code>	Преобразует все буквы в интервале между <code>beg</code> и <code>end</code> ; каждая буква заменяется результатом вызова <code>toupper()</code>
<code>ct.tolower(c)</code>	Возвращает букву нижнего регистра, соответствующую символу <code>c</code> . Если такой буквы не существует, возвращает <code>c</code>
<code>ct.tolower(beg, end)</code>	Преобразует все буквы в интервале между <code>beg</code> и <code>end</code> ; каждая буква заменяется результатом вызова <code>tolower()</code>

продолжение ↓

Таблица 14.16 (продолжение)

Выражение	Описание
<code>ct.widen(c)</code>	Возвращает символ <code>char</code> <i>c</i> , преобразованный к типу <code>charT</code>
<code>ct.widen(beg, end, dest)</code>	Для каждого символа в интервале между <i>beg</i> и <i>end</i> помещает результат вызова <code>widen()</code> в соответствующую позицию <i>dest</i>
<code>ct.narrow(c, default)</code>	Возвращает символ <code>charT</code> <i>c</i> , преобразованный к типу <code>char</code> , или символ <i>default</i> , если такого символа не существует
<code>ct.narrow(beg, end, default, dest)</code>	Для каждого символа в интервале между <i>beg</i> и <i>end</i> помещает результат вызова <code>narrow()</code> в соответствующую позицию <i>dest</i>

Функция `is(beg,end,vec)` сохраняет в массиве маски символов. Для каждого символа в интервале между *beg* и *end* в массиве, на который ссылается аргумент *vec*, сохраняется маска с установкой соответствующих атрибутов символов. Использование этой функции позволяет избежать многократных вызовов виртуальных функций при большом количестве классифицируемых символов.

Функция `widen()` преобразует символ типа `char` из исходной кодировки в кодировку, используемую локальным контекстом. Это означает, что вызов `widen()` может быть оправдан даже в том случае, если результат тоже относится к типу `char`. Обратное преобразование осуществляется функцией `narrow()`, которая преобразует символ из кодировки, используемой локальным контекстом, в соответствующий символ `char` (при условии, что такой символ существует). В следующем фрагменте строка с десятичными цифрами преобразуется из типа `char` в `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);
```

Класс `ctype` является производным от класса `ctype_base`. Этот класс используется только для определения перечисляемого типа `mask`. В перечисляемом типе задаются значения, сочетание которых образует битовую маску для проверки свойств символов. Значения, определенные в классе `ctype_base`, перечислены в табл. 14.17. Функциям классификации символов передается битовая маска, которая представляет собой комбинацию этих значений с поразрядными операторами `|`, `&`, `^` и `~`. Символ совпадает с маской только в том случае, если он принадлежит хотя бы к одной категории, определяемой этой маской.

Таблица 14.17. Маски символов, используемые фасетом `ctype`

Выражение	Описание
<code>ctype_base::alnum</code>	Буквы и цифры (эквивалент <code>alpha digit</code>)
<code>ctype_base::alpha</code>	Буквы
<code>ctype_base::cntrl</code>	Управляющие символы

Выражение	Описание
ctype_base::digit	Десятичные цифры
ctype_base::graph	Знаки препинания, буквы и цифры (эквивалент alnum punct)
ctype_base::lower	Буквы нижнего регистра
ctype_base::print	Печатные символы
ctype_base::punct	Знаки препинания
ctype_base::space	Пропуски
ctype_base::upper	Буквы верхнего регистра
ctype_base::xdigit	Шестнадцатеричные цифры

Специализации ctype для типа char

Чтобы повысить эффективность работы функций классификации символов, фацет ctype специализируется для типа символов `char`. Специализация не перепоручает функции классификации символов (`is()`, `scan()` и `scan_not()`) соответствующим виртуальным функциям. Вместо этого задействуется непосредственная реализация функций с использованием поиска по таблице категорий, для чего в фацете определяются дополнительные члены (табл. 14.18).

Таблица 14.18. Дополнительные члены ctype<char>

Выражение	Описание
ctype<char>::table_size	Размер таблицы (>=256)
ctype<char>::classic_table()	Возвращает таблицу категорий для «классического» локального контекста С
ctype<char>(table, del=false)	Создает фацет с таблицей категорий table
ct.table()	Возвращает текущую таблицу категорий для фацета ct

Чтобы изменить поведение этих функций для конкретного локального контекста, передайте соответствующую таблицу категорий в аргументе конструктора:

```
// Создание и инициализация таблицы
std::ctype_base::mask mytable[std::ctype<char>::table_size] = {
    ...
};

// Использовать таблицу для фацета ct типа ctype<char>
std::ctype<char> ct(mytable, false);
```

Фрагмент конструирует фацет `ctype<char>`, который классифицирует символы по таблице `mytable`. Точнее говоря, класс символа с определяется выражением:

```
mytable[static_cast<unsigned char>(c)]
```

Статическая переменная `table_size` является константой, определенной реализацией библиотеки и хранящей размер таблицы категорий. 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 при вызове `delete[]`, когда фацет становится ненужным.

Защищенная функция `table()` возвращает таблицу, переданную в первом аргументе конструктора. Статическая защищенная функция `classic_table()` возвращает таблицу, используемую для классификации символов в классическом локальном контексте С.

Глобальные вспомогательные функции классификации символов

Для удобства работы с фацетом `ctype` определен ряд глобальных функций, перечисленных в табл. 14.9.

Таблица 14.19. Глобальные вспомогательные функции классификации символов

Функция	Описание
<code>isalnum(c, loc)</code>	Проверяет, является ли с буквой или цифрой (эквивалент <code>isalpha()&&isdigit()</code>)
<code>isalpha(c, loc)</code>	Проверяет, является ли с буквой
<code>iscntrl(c, loc)</code>	Проверяет, является ли с управляемым символом
<code>isdigit(c, loc)</code>	Проверяет, является ли с цифрой
<code>isgraph(c, loc)</code>	Проверяет, является ли с печатным символом, исключая пробелы (эквивалент <code>isalnum()&&ispunct()</code>)
<code>islower(c, loc)</code>	Проверяет, является ли с буквой нижнего регистра
<code>isprint(c, loc)</code>	Проверяет, является ли с печатным символом (включая пропуски)
<code>ispunct(c, loc)</code>	Проверяет, является ли с знаком препинания (то есть печатным символом, отличным от пробела, цифры или буквы)
<code>isspace(c, loc)</code>	Проверяет, является ли с пробелом
<code>isupper(c, loc)</code>	Проверяет, является ли с буквой верхнего регистра
<code>isxdigit(c, loc)</code>	Проверяет, является ли с шестнадцатеричной цифрой
<code>tolower(c, loc)</code>	Преобразует с из верхнего регистра в нижний
<code>toupper(c, loc)</code>	Преобразует с из нижнего регистра в верхний

Например, следующее выражение проверяет, является ли символ `c` буквой нижнего регистра в локальном контексте `loc`:

```
std::islower(c, loc)
```

Функция возвращает значение типа `bool`.

Следующее выражение возвращает символ `c`, преобразованный в букву верхнего регистра в локальном контексте `loc`:

```
std::toupper(c, loc)
```

Если `c` не является буквой нижнего регистра, функция возвращает первый аргумент без изменений.

Представленный ниже вызов и следующая за ним конструкция эквивалентны:

```
std::islower(c, loc)
std::use_facet<std::ctype<char>>(loc).is(std::ctype_base::lower, c)
```

В приведенном выражении вызывается функция `is()` фацета `ctype<char>`. Функция `is()` проверяет, относится ли символ `c` к какой-либо из категорий, определяемых маской (первый аргумент). Значения флагов маски определяются в классе `ctype_base`. Примеры использования вспомогательных функций приведены на с. 484 и 643.

Глобальные вспомогательные функции классификации символов соответствуют одноименным функциям С, получающим только один аргумент. Эти функции определяются в заголовочных файлах `<cctype>` и `<ctype.h>` и всегда используют текущий единый локальный контекст С¹. Работать с ними еще удобнее:

```
if (std::isdigit(c)) {
    ...
}
```

Тем не менее при использовании этих функций вы не сможете работать с разными локальными контекстами в одной программе, а также задействовать функции С с пользовательским фацетом `ctype`. Пример применения функций С для преобразования всех символов строки к верхнему регистру приведен на с. 481

Вспомогательные функции C++ не должны использоваться в тех фрагментах кода, которые критичны по быстродействию. Гораздо быстрее получить соответствующий фацет от локального контекста и работать с функциями этого объекта напрямую. Если требуется классифицировать большое количество символов в одном локальном контексте, имеется еще более эффективное решение (во всяком случае, для символов, не относящихся к типу `char`). Для классификации типичных символов можно воспользоваться функцией `is(beg, end, vec)`: эта функция строит для каждого символа из интервала `[beg, end]` маску с описанием свойств этого символа. Полученная маска сохраняется в элементе вектора `vec`, позиция которого соответствует позиции символа. Далее полученный вектор используется для быстрой идентификации символов.

Преобразование кодировок

Фацет `codecvt` выполняет преобразования между внутренней и внешней кодировками символов. Например, с его помощью можно преобразовывать символы из Unicode в EUC (Extended UNIX Code) при условии, что реализация стандартной библиотеки C++ поддерживает соответствующий фацет.

¹ Этот локальный контекст идентичен глобально назначенному локальному контексту C++ только в том случае, если последний вызов `locale::global()` производился с именованным контекстом и функция `setlocale()` с тех пор не вызывалась. В противном случае локальный контекст, используемый функциями С, будет отличаться от глобально назначенного локального контекста C++.

Этот фасет используется классом `basic_filebuf` для преобразования между внутренней кодировкой и представлением, хранящимся в файле. Класс `basic_filebuf<charT,traits>` (см. с. 602) задействует для этой цели специализацию `codecvt<charT,char,typename traits::state_type>`. Применяемый фасет берется из локального контекста, связанного с `basic_filebuf`. Данный способ применения фасета является основным, напрямую с этим фасетом работают крайне редко.

На с. 658 были приведены начальные сведения о кодировке символов. Чтобы понять, как работает фасет `codecvt`, необходимо знать о существовании двух схем кодировки: в одной схеме каждый символ кодируется фиксированным количеством байтов (расширенная кодировка), а в другой используется переменное количество байтов на символ (многобайтовая кодировка).

Также необходимо знать о том, что в многобайтовых кодировках для повышения эффективности представления используется так называемое *состояние сдвига*. Чтобы правильно интерпретировать байт, необходимо знать состояние сдвига для данной позиции. Состояние сдвига определяется только полным перебором всей последовательности многобайтовых символов (за подробностями обращайтесь к с. 658).

Шаблон фасета `codecvt<>` получает три аргумента:

- тип символов внутреннего представления `internT`;
- тип символов внешнего представления `charT`;
- тип `stateT`, используемый для представления промежуточного состояния в процессе преобразования.

Промежуточное состояние может состоять из незавершенных символов в расширенной кодировке или текущего состояния сдвига. Стандарт C++ не делает никаких ограничений по поводу того, что должно храниться в объектах, представляющих состояние.

Во внутреннем представлении символы всегда кодируются фиксированным количеством байтов. Как правило, программы работают с символьными данными типов `char` и `wchar_t`, а во внешнем представлении может использоваться многобайтовая или расширенная кодировка. В случае многобайтовой кодировки второй аргумент шаблона определяет тип представления ее базовых единиц. Каждый многобайтовый символ состоит из одного или нескольких объектов этого типа. Обычно для этой цели применяется тип `char`.

В третьем аргументе передается тип, используемый для представления текущего состояния преобразования. Например, если одна из кодировок является многобайтовой, обработка многобайтового символа может быть прервана из-за переполнения исходного или приемного буфера. В этом случае текущее состояние преобразования сохраняется в объекте заданного типа.

По аналогии с другими фасетами стандарт требует обязательной поддержки минимального количества специализаций. Стандартная библиотека C++ поддерживает только две специализации:

- `codecvt<char,char,mbstate_t>` — преобразование исходной кодировки самой в себя (вырожденная версия фасета `codecvt`);
- `codecvt<wchar_t,char,mbstate_t>` — преобразование между исходной «узкой» кодировкой (то есть `char`) и расширенной кодировкой (`wchar_t`).

Стандарт C++ не определяет конкретную семантику второго преобразования. Единственное естественное решение заключается в разбиении каждого объекта `wchar_t` на `sizeof(wchar_t)` объектов типа `char` для преобразования `wchar_t` в `char` и сборке `wchar_t` из того же количества объектов `char` при обратном преобразовании. Обратите внимание на принципиальные отличия этого преобразования от тех, которые выполняются функциями `widen()` и `narrow()` фасета `ctype`: если функции `codecvt` используют биты нескольких объектов `char` для формирования одного объекта `wchar_t` (или наоборот), функции `ctype` преобразуют символ из одной кодировки в соответствующий символ другой кодировки (если он существует).

Фасет `codecvt`, как и `ctype`, является производным от базового класса с определением перечисляемого типа. Базовый класс называется `codecvt_base`, в нем определяется перечисляемый тип `result`. Значения перечисляемого типа требуются при описании результатов вызова функций `codecvt`. Точный смысл каждого значения зависит от вызываемой функции. В табл. 14.20 перечислены функции фасета `codecvt`.

Таблица 14.20. Функции фасета `codecvt`

Выражение	Описание
<code>cvt.in(s, fb, fe, fn, tb, te, tn)</code>	Преобразует внешнее представление к внутреннему
<code>cvt.out(s, fb, fe, fn, tb, te, tn)</code>	Преобразует внутреннее представление к внешнему
<code>cvt.unshift(s, tb, te, tn)</code>	Записывает служебную последовательность для переключения к исходному состоянию сдвига
<code>cvt.encoding()</code>	Возвращает информацию о внешней кодировке
<code>cvt.always_noconv()</code>	Возвращает <code>true</code> , если преобразование не выполняется
<code>cvt.length(s, fb, fe, max)</code>	Возвращает количество объектов <code>externT</code> в интервале между <code>fb</code> и <code>fe</code> для получения <code>max</code> символов во внутреннем представлении
<code>cvt.max_length()</code>	Возвращает максимальное количество объектов <code>externT</code> , необходимых для создания одного объекта <code>internT</code>

Функция `in()` преобразует внешнее представление к внутреннему. Аргумент `s` содержит ссылку на `stateT`. Перед преобразованием этот аргумент определяет состояние сдвига, используемое в начале преобразования, а после преобразования в нем сохраняется итоговое состояние сдвига. Переданное состояние сдвига может отличаться от исходного, если текущий буфер не является первым из обрабатываемых буферов. Аргументы `fb` и `fe` относятся к типу `const internT*` и определяют соответственно начало и конец входного буфера. Аргументы `tb` и `te` относятся к типу `externT*` и определяют соответственно начало и конец выходного буфера. Аргументы `fn` (тип `const externT*&`) и `tn` (тип `internT*&`) содержат ссылки, в которых возвращается конец преобразуемой последовательности символов во входном и выходном буферах соответственно. Конец любого буфера может быть достигнут до того, как будет достигнут конец другого буфера. Функция возвращает значение типа `codecvt_base::result` (табл. 14.21).

Таблица 14.21. Возвращаемые значения функций преобразования

Выражение	Описание
ok	Все исходные символы были успешно преобразованы
partial	Не все символы были успешно преобразованы, или для получения преобразованного символа нужны дополнительные символы
error	Обнаружен исходный символ, который невозможно преобразовать
посопн	Преобразование не требуется

Возвращаемое значение `ok` означает, что функция успешно справилась со своей работой. Если $fn == fe$, это означает, что входной буфер был обработан полностью, а последовательность в интервале между tb и tn содержит результат преобразования. Символы этой последовательности соответствуют символам входной последовательности (возможно — с добавлением законченного символа от предыдущего преобразования). Если аргумент `s` функции `in()` не находился в исходном состоянии, возможно, в нем хранился незавершенный символ от предыдущего преобразования.

Если функция возвращает значение `partial`, это может означать одно из двух: либо выходной буфер был заполнен до завершения чтения входного буфера, либо входной буфер был прочитан полностью при наличии незавершенных символов (например, если последний байт входной последовательности был частью служебной последовательности для переключения состояний сдвига). Если $fe == fn$, входной буфер был прочитан полностью. В этом случае последовательность между tb и tn содержит все полностью преобразованные символы. Информация, необходимая для завершения преобразования символа, сохраняется в состоянии сдвига `s`. Если $fe != fn$, входной буфер был прочитан не полностью. В этом случае следующее преобразование будет продолжено с позиции `fn`.

Возвращаемое значение `посопн` является признаком особой ситуации: оно означает, что преобразования внешнего представления во внутреннее не потребовалось. В этом случае `fn` присваивается `fb`, а `tn` присваивается `tb`. В приемной последовательности не сохраняется ничего, поскольку вся необходимая информация уже хранится во входной последовательности.

Если функция возвращает код `error`, это означает, что исходный символ преобразовать не удалось. Это может произойти по нескольким причинам: например, в приемной кодировке отсутствует 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 `in()`, но она работает в обратном направлении, то есть преобразует внутреннее представление во внешнее. Аргументы и возвращаемые значения двух функций совпадают, различаются только типы аргументов: `tb` и `te` относятся к типу `const internT*`, а `fb` и `fe` — к типу `const externT*`. То же можно сказать и об аргументах `fn` и `tn`.

Функция `unshift()` вставляет символы, необходимые для завершения последовательности, когда в аргументе `z` передается текущее состояние преобразования. Обычно это означает, что состояние сдвига возвращается к исходному. Завершается только внешнее представление. Аргументы `tb` и `tf` относятся к типу `externT*`, а аргумент `tn` – к типу `externT&*`. Интервал между `tb` и `te` определяет выходной буфер, в котором сохраняются символы. Конец выходной последовательности хранится в `tn`. Значения, возвращаемые функций `unshift()`, перечислены в табл. 14.22.

Таблица 14.22. Возвращаемые значения функций преобразования

Выражение	Описание
<code>ok</code>	Последовательность завершена успешно
<code>partial</code>	Для завершения последовательности нужны дополнительные символы
<code>error</code>	Недействительное состояние
<code>nosopv</code>	Для завершения последовательности символы не требуются

Функция `encoding()` возвращает дополнительную информацию о кодировке внешнего представления. Если `encoding()` возвращает `-1`, преобразование зависит от состояния. Если `encoding()` возвращает `0`, то количество объектов `externT`, необходимых для построения символа во внутренней кодировке, является переменной величиной. В противном случае функция возвращает количество объектов `externT`, необходимых для построения `internT`. Эта информация может использоваться для выделения буфера соответствующего размера.

Функция `always_nosopv()` возвращает `true`, если функции `in()` и `out()` никогда не выполняют преобразования. Например, стандартная реализация `codecvt<char, char, mbstate_t>` преобразований не выполняет, поэтому для этого фасета функция `always_nosopv()` всегда возвращает `true`. Тем не менее это относится только к фасету `codecvt` локального контекста "C", другие экземпляры этого фасета могут выполнять преобразования.

Функция `length()` возвращает количество объектов `externT` в интервале между `fb` и `fe`, необходимых для построения `max` символов типа `internT`. Если интервал между `fb` и `fe` содержит менее `max` полных символов типа `internT`, возвращается количество объектов `externT`, необходимых для построения максимально возможного количества символов типа `internT`.

Контекстная сортировка

Фасет `collate` скрывает различия между правилами сортировки строк в разных локальных контекстах. Например, в немецком языке буква «ÿ» при сортировке строк эквивалентна букве «и» или сочетанию «ие». В других языках эта буква даже не считается буквой и интерпретируется как специальный символ (или не интерпретируется вовсе), но в этих языках действуют иные правила сортировки для некоторых сочетаний символов. Чтобы строки автоматически сортировались

в порядке, нужном для пользователя, можно задействовать фацет `collate`. Функции этого фацета перечислены в табл. 14.23. `Col` обозначает специализацию `collate`, а в аргументах функций передаются итераторы, используемые для определения строк.

Таблица 14.23. Функции фацета `collate`

Выражение	Описание
<code>col.compare(beg1, end1, beg2, end2)</code>	Возвращаемое значение равно 1, если первая строка больше второй; 0, если строки равны; -1, если первая строка меньше второй
<code>col.transform(beg, end)</code>	Возвращает строку, предназначенную для сравнения с другими преобразованными строками
<code>col.hash(beg, end)</code>	Возвращает хэш-код строки (типа <code>long</code>)

Фацет `collate` представляет собой шаблон класса, получающий в аргументе шаблона тип символов `charT`. Строки, передаваемые функциям `collate`, задаются при помощи указателей `const charT*`. Это несколько неудобно, поскольку нет гарантии, что итераторы, используемые типом `basic_string<charT>`, также представляют собой указатели. Следовательно, сравнение строк должно производиться фрагментами следующего вида:

```
locale loc;
string s1, s2;
...
// Получение фацета collate локального контекста locale
const std::collate<charT>& col
    = std::use_facet<std::collate<charT>>(loc);

// Сравнение строк с использованием фацета collate
int result = col.compare(s1.data(), s1.data() + s1.size(),
                         s2.data(), s2.data() + s2.size());
if (result == 0) {
    // s1 и s2 равны
}
...
```

Такие ограничения связаны с тем, что нужный тип итератора заранее не известен, а определять фацеты для типа указателя и бесконечного числа типов итераторов невозможно.

Конечно, для сравнения строк в классе `locale` существует специальная вспомогательная функция:

```
int result = loc(s1, s2);
```

Однако такое решение работает только с функцией `compare()`. В стандартной библиотеке C++ не определены вспомогательные функции для двух других функций `collate`.

Функция `transform()` возвращает объект типа `basic_string<charT>`. Лексикографический порядок строк, возвращаемых функцией `transform()`, совпадает с порядком следования исходных строк, сортируемых функцией `collate()`. Функция `transform()` может использоваться для ускорения работы программы, если одна строка сравнивается с большим количеством других строк. Лексикографический порядок следования строк в этом случае определяется гораздо быстрее, чем при использовании `collate()`, поскольку национальные правила сортировки могут быть довольно сложными.

Стандартная библиотека C++ требует обязательной поддержки только двух специализаций — `collate<char>` и `collate<wchar_t>`. Для других типов символов пользователи должны писать собственные специализации, в которых могут быть задействованы стандартные специализации.

Интернационализация сообщений

Фацет `messages` предназначен для выборки интернационализированных сообщений из каталога. В основном он используется для предоставления сервиса, аналогичного функции `perror()`. В POSIX-совместимых системах эта функция выводит системное сообщение об ошибке, номер которой хранится в глобальной переменной `errno`. Фацет `messages` гораздо более универсален. К сожалению, в спецификации он определен недостаточно четко.

Фацет `messages` представляет собой шаблон класса, получающий в аргументе шаблона тип символов `charT`. Строки, возвращаемые этим фацетом, относятся к типу `basic_string<charT>`. Базовая схема использования шаблона основана на открытии каталога, чтении сообщений и закрытии каталога. Класс `messages` объявлен производным от класса `messages_base`, в котором определяется тип `catalog` (в действительности это определение типа для `int`). Объект типа `catalog` идентифицирует каталог, с которым работают функции фацета `messages`. В табл. 14.24 приведен список этих функций.

Таблица 14.24. Функции фацета `messages`

Выражение	Описание
<code>msg.open(name, loc)</code>	Открывает каталог и возвращает соответствующий идентификатор
<code>msg.get(cat, set, msgid, def)</code>	Возвращает сообщение с идентификатором <code>msgid</code> из каталога <code>cat</code> ; если такое сообщение отсутствует, возвращает <code>def</code>
<code>msg.close(cat)</code>	Закрывает каталог <code>cat</code>

Имя, передаваемое в аргументе 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 фацу `ctype`. Фацет обеспечивает преобразование сообщений к нужному типу символов.

Точная семантика функции `get()` не определена. Например, реализация для POSIX-совместимой системы может вернуть строку, соответствующую сообщению об ошибке `msgid`, но такое поведение не является обязательным по требованиям стандарта. Аргумент `set` предназначен для дополнительного структурирования сообщений, например, чтобы различать системные ошибки и ошибки стандартной библиотеки C++.

После выполнения необходимых операций каталог сообщений освобождается функцией `close()`. Хотя интерфейс `open()/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 `close()` освободит эту память.

Стандарт требует, чтобы в каждом локальном контексте хранились две специализации — `messages<char>` и `messages<wchar_t>`. Другие специализации стандартной библиотеки C++ не поддерживаются.

15 Распределители памяти

Распределители памяти были рассмотрены на с. 49. Они представляют собой специализированные абстракции, преобразующие *потребности* в памяти в не-посредственные *вызовы*. Принципы работы распределителей памяти (далее – просто «распределителей») подробно рассматриваются в этой главе.

Использование распределителей в прикладном программировании

С точки зрения прикладного программиста, работать с разными распределителями памяти несложно; достаточно передать нужный распределитель в аргументе шаблона. Например, следующий фрагмент создает разные контейнеры и строки с использованием специального распределителя `MyAlloc<>`:

```
// Вектор со специальным распределителем
std::vector<int, MyAlloc<int> > v;

// Отображение int/float со специальным распределителем
std::map<int, float, less<int>,
         MyAlloc<std::pair<const int, float> > > m;

// Стока со специальным распределителем
std::basic_string<char, std::char_traits<char>, MyAlloc<char> > s;
```

Если вы используете собственный распределитель памяти, вероятно, для него стоит определить некоторые типы. Пример:

```
// Специальный строковый тип, использующий специальный распределитель
typedef basic_string<char, std::char_traits<char>,
                     MyAlloc<char> > xstring;

// Специальное отображение string/string со специальным распределителем
typedef std::map<xstring, xstring, less<xstring>,
             MyAlloc<std::pair<const xstring, xstring> >> xmap;

// Создание объекта типа xmap
xmap xmap;
```

Объекты с нестандартными распределителями памяти внешне ничем не отличаются от обычных объектов. Однако нельзя забывать, что элементы с разными распределителями не должны смешиваться; в противном случае возможны непредсказуемые последствия. Чтобы проверить, используют ли два распределителя одну модель памяти, воспользуйтесь оператором `==`. Если оператор возвращает `true`, то память, выделенная одним распределителем, может быть освобождена через другой распределитель. Во всех типах, параметризованных по распределителю, определена функция `get_allocator()`, предназначенная для получения объекта распределителя. Пример:

```
if (tumap.get_allocator() == s.get_allocator()) {
    // OK, tumap и s используют одинаковые
    // или взаимозаменяемые распределители
    ...
}
```

Использование распределителей при программировании библиотек

Этот раздел посвящен использованию распределителей с точки зрения людей, занимающихся реализацией контейнеров и других компонентов, способных работать с разными распределителями. Материал этого раздела частично основан на разделе 19.4 книги Баярна Страуструпа «*The C++ Programming Language*», 3rd Edition (с разрешения автора).

Распределители предоставляют интерфейс для выделения и освобождения памяти, создания и уничтожения объектов (табл. 15.1). Распределители делают возможной параметризацию контейнеров и алгоритмов по способу хранения элементов. Например, они могут работать с общей памятью или отображать элементы на записи базы данных.

Таблица 15.1. Основные операции распределителей

Выражение	Описание
<code>a.allocate(num)</code>	Выделение памяти для num элементов
<code>a.construct(p)</code>	Инициализация элемента, на который ссылается указатель p
<code>a.destroy(p)</code>	Уничтожение элемента, на который ссылается указатель p
<code>a.deallocate(p, num)</code>	Освобождение памяти для num элементов, на которую ссылается указатель p

В качестве примера рассмотрим упрощенную реализацию вектора. Вектор получает распределитель памяти в виде аргумента шаблона или конструктора и сохраняет ее в своем внутреннем представлении:

```
namespace std {
    template <class T,
              class Allocator = allocator<T> >
```

```

class vector {
    ...
private:
    Allocator alloc;      // Распределитель
    T*     elems;        // Массив элементов
    size_type numElems; // Количество элементов
    size_type sizeElems; // Объем памяти для хранения элементов
    ...

public:
    // Конструкторы
    explicit vector(const Allocator& = Allocator());
    explicit vector(size_type num, const T& val = T(),
                    const Allocator& = Allocator());
    template <class InputIterator>
    vector(InputIterator beg, InputIterator end,
           const Allocator& = Allocator());
    vector(const vector<T,Allocator>& v);
};

}

```

Возможная реализация второго конструктора, инициализирующего вектор num элементами со значением val, может выглядеть так:

```

namespace std {
    template <class T, class Allocator>
    vector<T,Allocator>::vector(size_type num, const T& val,
                                const Allocator& a)
        : alloc(a)          // Инициализация распределителя
    {
        // Выделение памяти
        sizeElems = numElems = num;
        elems = alloc.allocate(num);

        // Инициализация элементов
        for (size_type i=0; i<num; ++i) {
            // Инициализация i-го элемента
            alloc.construct(&elems[i], val);
        }
    }
}

```

В стандартную библиотеку C++ включены также вспомогательные функции для инициализации неинициализированной памяти (табл. 15.2). При использовании этих функций реализация конструктора выглядит еще проще:

```

namespace std {
    template <class T, class Allocator>
    vector<T,Allocator>::vector(size_type num, const T& val,
                                const Allocator& a)
        : alloc(a)          // Инициализация распределителя

```

```

{
    // Выделение памяти
    sizeElements = numElements = num;
    elems = alloc.allocate(num);

    // Инициализация элементов
    uninitialized_fill_n (elems, num, val);
}
}

```

Таблица 15.2. Вспомогательные функции для инициализации неинициализированной памяти

Выражение	Описание
uninitialized_fill(beg, end, val)	Инициализация интервала [beg,end) значением val
uninitialized_fill_n(beg, num, val)	Инициализация num элементов, начиная с beg, значением val
uninitialized_copy(beg, end, mem)	Инициализация элементов, начиная с mem, элементами интервала [beg,end)

Функция `reserve()`, которая резервирует дополнительную память без изменения количества элементов (см. с. 158), может быть реализована так:

```

namespace std {
    template <class T, class Allocator>
    void vector<T,Allocator>::reserve(size_type size)
    {
        // Функция reserve() никогда не уменьшает объем памяти
        if (size <= sizeElements) {
            return;
        }

        // Выделение новой памяти для size элементов
        T* newmem = alloc.allocate(size);

        // Копирование старых элементов в новую память
        uninitialized_copy(elems,elems+numElements,newmem);

        // Уничтожение старых элементов
        for (size_type i=0; i<numElements; ++i) {
            alloc.destroy(&elems[i]);
        }

        // Освобождение старой памяти
        alloc.deallocate(elems,sizeElements);

        // Элементы находятся в новой памяти
        sizeElements = size;
        elems = newmem;
    }
}

```

Инициализирующий итератор

Класс `raw_storage_iterator` предназначен для перебора неинициализированной памяти и ее инициализации. Итератор `raw_storage_iterator` может использовать любые алгоритмы для инициализации памяти значениями, полученными в результате выполнения алгоритма.

Например, следующая команда 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)); // Приемник
```

В первом аргументе шаблона (в данном случае `T*`) должен передаваться итератор вывода для типа элементов. Второй аргумент шаблона (в данном случае `T`) определяет тип элементов.

Временные буфера

Функции `get_temporary_buffer()` и `return_temporary_buffer()` работают с неинициализированной памятью, предназначенней для краткосрочного использования внутри функции. Учтите, что функция `get_temporary_buffer()` может вернуть меньше памяти, чем ожидалось, поэтому она возвращает пару из адреса и размера памяти (в элементах). Пример использования этой функции:

```
void f()
{
    // Выделение памяти для num элементов типа MyType
    pair<MyType*,std::ptrdiff_t> p
    = get_temporary_buffer<MyType>(num);

    if (p.second == 0) {
        // Выделить память для элементов вообще не удалось
        ...
    }
    else if (p.second < num) {
        // Не удалось выделить достаточно памяти для num элементов.
        // Однако и эту память нужно освободить!
        ...
    }

    // Обработка
    ...

    // Освобождение временной выделенной памяти
    if (p.first != 0) {
        return_temporary_buffer(p.first);
    }
}
```

Функции `get_temporary_buffer()` и `return_temporary_buffer()` слишком сложны для написания безопасного с точки зрения исключений кода, поэтому обычно они не встречаются в библиотечных реализациях.

Распределитель по умолчанию

Распределитель по умолчанию объявляется следующим образом:

```
namespace std {
    template <class 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;

            // Привязка распределителя к типу U
            template <class U>
            struct rebind {
                typedef allocator<U> other;
            };

            // Функции возвращают адреса значений
            pointer      address(reference value) const;
            const_pointer address(const_reference value) const;

            // Конструкторы и деструктор
            allocator() throw();
            allocator(const allocator&) throw();
            template <class U>
            allocator(const allocator<U>&) throw();
            ~allocator() throw();

            // Функция возвращает максимальное количество элементов,
            // для которых может быть выделена память
            size_type max_size() const throw();

            // Выделение памяти для num элементов типа T без инициализации
            pointer allocate(size_type num,
                             allocator<void>::const_pointer hint=0);

            // Инициализация элементов выделенного блока р значением value
            void construct(pointer p, const T& value);
    };
}
```

```

// Удаление элементов инициализированного блока p
void destroy(pointer p);

// Освобождение блока p с удаленными элементами
void deallocate(pointer p, size_type num);
};

}

}

```

Распределитель по умолчанию выделяет и освобождает память глобальными операторами `new` и `delete`. Это означает, что вызов `allocate()` может генерировать исключение `bad_alloc`. Однако распределитель по умолчанию можно оптимизировать за счет повторного использования освобожденной памяти или выделения лишней памяти для экономии времени на дополнительных перераспределениях. Следовательно, моменты вызова операторов `new` и `delete` точно не известны. Возможная реализация пользовательского распределителя приведена на с. 708.

Внутри шаблона распределителя имеется несколько необычное определение шаблонной структуры `rebind`. Эта шаблонная структура дает возможность косвенного выделения памяти другого типа. Например, если распределитель относится к типу `Allocator`, то следующая конструкция определяет тип того же распределителя, специализированного для элементов типа `T2`:

```
Allocator::rebind<T2>::other
```

Представьте, что при реализации контейнера вам потребовалось выделить память для типа, отличного от типа элементов. Например, при реализации дека обычно приходится выделять память для массивов, управляющих блоками элементов (типичная реализация дека представлена на с. 169). Следовательно, вам понадобится распределитель для выделения массивов указателей на элементы:

```

namespace std {
    template <class T,
              class Allocator = allocator<T>>
    class deque {
        ...
        private:
            // Привязка распределителя к типу T*
            typedef typename Allocator::rebind<T*>::other PtrAllocator;

            Allocator    alloc;          // Распределитель для значений типа T
            PtrAllocator block_alloc;   // Распределитель для значений типа T*
            T**          elems;         // Массив блоков элементов
        ...
    };
}

```

При управлении элементами дека один распределитель должен работать с блоками элементов, а другой — с массивом блоков элементов. Последний относится к типу `PtrAllocator`. При помощи структуры `rebind<>` распределитель элементов (`Allocator`) привязывается к типу массива элементов (`T*`).

Распределитель 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 <class U>
            struct rebind {
                typedef allocator<U> other;
            };
    };
}
```

Пользовательский распределитель

Написать собственный распределитель не так уж сложно. Самый важный вопрос — как будет организовано выделение или освобождение памяти? Все остальное более или менее очевидно. Для примера рассмотрим упрощенную реализацию распределителя по умолчанию:

```
// util/defalloc.hpp
namespace std {
    template <class 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;

            // Привязка распределителя к типу U
            template <class U>
            struct rebind {
                typedef allocator<U> other;
            };

            // Функции возвращают адреса значений
            pointer address (reference value) const {
                return &value;
            }
    };
}
```

```
const_pointer address (const_reference value) const {
    return &value;
}

/* Конструкторы и деструктор
 * - ничего не делаем, поскольку распределитель не имеет состояния
 */
allocator() throw() {
}
allocator(const allocator&) throw() {
}
template <class U>
allocator (const allocator<U>&) throw() {
}
~allocator() throw() {
}

// Функция возвращает максимальное количество элементов,
// для которых может быть выделена память
size_type max_size () const throw() {
    return numeric_limits<size_t>::max() / sizeof(T);
}

// Выделение памяти для num элементов типа T без инициализации
pointer allocate (size_type num,
                   allocator<void>::const_pointer hint = 0) {
    // allocate memory with global new
    return (pointer)(::operator new(num*sizeof(T)));
}

// Инициализация элементов выделенного блока p значением value
void construct (pointer p, const T& value) {
    // Инициализация памяти оператором new
    new((void*)p)T(value);
}

// Удаление элементов инициализированного блока p
void destroy (pointer p) {
    // Уничтожение объектов вызовом деструктора
    p->~T();
}

// Освобождение блока p, содержащего удаленные элементы
void deallocate (pointer p, size_type num) {
    // Освобождение памяти глобальным оператором delete
    ::operator delete((void*)p);
}
};
```

```
// Оператор сообщает, что все специализации данного распределителя
// являются взаимозаменяемыми.
template <class T1, class T2>
bool operator==(const allocator<T1>&,
                  const allocator<T2>&) throw() {
    return true;
}
template <class T1, class T2>
bool operator!=(const allocator<T1>&,
                  const allocator<T2>&) throw() {
    return false;
}
```

Используя эту базовую реализацию, вы можете без особых трудностей реализовать собственный распределитель. Как правило, все отличия от приведенной реализации сосредоточены в функциях `max_size()`, `allocate()` и `deallocate()`. В этих трех функциях вы программируете политику выделения памяти (например, повторного использования памяти вместо ее немедленного освобождения), расходования общей памяти или отображения памяти на сегменты объектно-ориентированной базы данных.

За дополнительными примерами обращайтесь по адресу <http://www.josuttis.com/libbook/examples.html>.

Требования к распределителям памяти

Ниже перечислены типы и операции, которые должны поддерживаться распределителями в соответствии с требованиями стандарта. Для распределителей, которые могут использоваться стандартными контейнерами, устанавливаются особые требования — более жесткие, чем для обычных распределителей.

Определения типов

`распределитель::value_type`

- Тип элементов.
- Эквивалент `T` для `allocator<T>`.

`распределитель::size_type`

- Тип беззнаковых целочисленных значений, представляющих размер наибольшего объекта в модели распределения памяти.
- Чтобы распределитель мог использоваться со стандартными контейнерами, этот тип должен быть эквивалентен `size_t`.

`распределитель::difference_type`

- Тип знаковых целочисленных значений, представляющих разность между двумя указателями в модели распределения памяти.

- Чтобы распределитель мог использоваться со стандартными контейнерами, этот тип должен быть эквивалентен `ptrdiff_t`.

распределитель::pointer

- Тип указателя на тип элемента.

- Чтобы распределитель мог использоваться со стандартными контейнерами, этот тип должен быть эквивалентен `T*` для `allocator<T>`.

распределитель::const_pointer

- Тип константного указателя на тип элемента.

- Чтобы распределитель мог использоваться со стандартными контейнерами, этот тип должен быть эквивалентен `const T*` для `allocator<T>`.

распределитель::reference

- Тип ссылки на тип элемента.

- Эквивалент `T&` для `allocator<T>`.

распределитель::const_reference

- Тип константной ссылки на тип элемента.

- Эквивалент `const T&` для `allocator<T>`.

распределитель::rebind

- Шаблонная структура, позволяющая любому распределителю косвенно выделять память для другого типа.

- Объявление должно выглядеть так:

```
template <class T>
class распределитель {
public:
    template <class U>
    struct rebind {
        typedef распределитель<U> other;
    };
    ...
}
```

- Назначение структуры `rebind` разъясняется на с. 707.

Операции

распределитель::allocator ()

- Конструктор по умолчанию.
- Создает объект распределителя.

распределитель::allocator (const allocator& a)

- Копирующий конструктор.

- Создает объект распределителя таким образом, что память, выделенная оригиналом, может освобождаться через копию, и наоборот.

распределитель::~allocator ()

- Деструктор.

- Уничтожает объект распределителя.

pointer распределитель::address (*reference value*)
const_pointer распределитель::address (*const_reference value*)

- Первая форма возвращает неконстантный указатель на неконстантное значение *value*.

- Вторая форма возвращает константный указатель на константное значение *value*.

распределитель::max_size ()

- Возвращает максимальное значение, которое может передаваться функции *allocate()* для выделения памяти.

pointer распределитель::allocate (*size_type num*)
const_pointer распределитель::allocate (*size_type num*,
allocator<void>::*const_pointer hint*)

- Обе формы возвращают указатель на память, выделенную для *num* элементов типа *T*.

- Элементы не конструируются и не инициализируются (для них не вызываются конструкторы).

- Смысл необязательного второго аргумента определяется реализацией. Например, в нем может передаваться дополнительная информация для повышения эффективности.

void распределитель::deallocate (*pointer p*, *size_type num*)

- Освобождает память, на которую ссылается указатель *p*.

- Блок *p* должен быть выделен функцией *allocate()* того же или равного распределителя.

- Аргумент *p* не может быть равен **NULL** или 0.

- К моменту вызова *deallocate()* элементы должны быть уже уничтожены.

void распределитель::construct (*pointer p*, *const T& value*)

- Инициализирует память одного элемента, на которую ссылается указатель *p*, значением *value*.

- Эквивалент:

new((void)p)T(value)*

void allocator::destroy (*pointer p*)

- Уничтожает объект, на который ссылается указатель *p*, без освобождения памяти.

- Просто вызывает деструктор объекта.
- Эквивалент:

```
((T*)p)->~T()
```

```
bool operator==(const распределитель& a1, const распределитель& a2)
```

- Возвращает `true`, если распределители *a1* и *a2* взаимозаменяемы.
- Два объекта распределителя считаются взаимозаменяемыми, если память, выделенная одним из них, может быть освобождена другим.
- Чтобы распределитель мог использоваться со стандартными контейнерами, распределители одного типа обязаны быть взаимозаменяемыми, поэтому функция должна всегда возвращать `true`.

```
bool operator!=(const распределитель& a1, const распределитель& a2)
```

- Возвращает `true`, если распределители *a1* и *a2* взаимозаменяемы.
- Эквивалент:

```
!(a1==a2)
```

- Чтобы распределитель мог использоваться со стандартными контейнерами, распределители одного типа обязаны быть взаимозаменяемыми, поэтому функция должна всегда возвращать `false`.

Операции с неинициализированной памятью

В этом разделе подробно описаны вспомогательные функции для работы с неинициализированной памятью. Примеры реализаций, безопасных с точки зрения исключений, основаны (с разрешения автора) на коде Грега Колвина (Greg Colvin).

```
void uninitialized_fill(ForwardIterator beg, ForwardIterator end,
                        const T& value)
```

- Инициализирует элементы интервала $[beg, end]$ значением *value*.
- Функция либо выполняется успешно, либо не вносит изменений.
- Типичная реализация `uninitialized_fill()`:

```
namespace std {
    template <class ForwIter, class 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;
}
}
}

void uninitialized_fill_n (ForwardIterator beg, Size num,
                           const T& value)

```

- Инициализирует *num* элементов, начиная с *beg*, значением *value*.
- Функция либо выполняется успешно, либо не вносит изменений.
- Типичная реализация `uninitialized_fill_n()`:

```

namespace std {
    template <class ForwIter, class Size, class T>
    void 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);
            }
        }
        catch (...) {
            for (; save!=beg; ++save) {
                save->~VT();
            }
            throw;
        }
    }
}

```

- Пример использования функции `uninitialized_fill_n()` приведен на с. 703.

```

void uninitialized_copy (InputIterator sourceBeg,
                        InputIterator sourceEnd,
                        ForwardIterator destBeg)

```

- Элементами из интервала $[sourceBeg, sourceEnd)$ инициализирует память, начиная с *destBeg*.
- Функция либо выполняется успешно, либо не вносит изменений.

○ Типичная реализация `uninitialized_copy()`:

```
namespace std {
    template <class InputIter, class ForwIter>
    ForwIter uninitialized_fill_n (InputIter beg, InputIter end,
                                  ForwIter dest)
    {
        typedef typename iterator_traits<ForwIter>::value_type VT;
        ForwIter save(dest);
        try {
            for (; beg!=end; ++beg,++dest) {
                new (static_cast<void*>(&*dest))VT(value);
            }
            return dest;
        }
        catch (...) {
            for (; save!=dest; ++save) {
                save->~VT();
            }
            throw;
        }
    }
}
```

Пример использования функции `uninitialized_copy()` приведен на с. 704

Ресурсы Интернета

В Интернете можно найти огромный объем дополнительной информации по темам, рассмотренным в книге. Ниже приводится список полезных сайтов.

Группы Usenet

Ниже перечислены группы Usenet, посвященные C++, стандарту и стандартной библиотеке C++.

- Общие аспекты C++ (не модерируется):

comp.lang.c++

- Общие аспекты C++ (модерируется):

comp.lang.c++.moderated

- Стандарт C++ (модерируется):

comp.std.c++

За дополнительной информацией обращайтесь на сайт:

<http://reality.sgi.com/austern/std-c++/faq.html>

URL-адреса

Здесь перечислены ссылки на источники дополнительной информации о стандартной библиотеке C++ и STL. Однако книги живут дольше интернет-сайтов, поэтому некоторые ссылки могут оказаться недействительными. Обновленный список ссылок можно загрузить с сайта книги (автор надеется, что этот сайт будет существовать достаточно долго):

<http://www.josuttis.com/libbook/>

Ниже перечислены ссылки, относящиеся к стандартной библиотеке C++ в целом.

- Список FAQ о стандартизации C++: ,

<http://reality.sgi.com/austern/std-c++/faq.html>

- Официальная страница рабочей группы ISO по стандартизации C++:
<http://www.dkuug.dk/jtc1/sc22/wg21/>
- Справочник Dinkum по библиотеке C++:
<http://www.dinkumware.com/refcpp.html>
- Реализация стандартной библиотеки C++ для компилятора EGCS C++:
<http://sourceware.cygnus.com/libstdc++/>
- Компилятор EGCS C++:
<http://egcs.cygnus.com/>
- Boost, архив бесплатно распространяемых библиотек C++:
<http://www.boost.org/>
- Blitz++, библиотека классов C++ для научных вычислений:
<http://www.oonumerics.org/blitz/>
Следующие ссылки посвящены общим вопросам STL.
- Бесплатная реализация STL от SGI:
<http://www.sgi.com/Technology/STL/>
- STLport для разных платформ:
<http://www.stlport.org/>
- Руководство по STL для начинающих:
<http://www.xraylith.wisc.edu/~khan/software/stl/STL.newbie.html>
- Сайт STL Дэвида Мюссера:
<http://www.cs.rpi.edu/~musser/stl.html>
- Списки часто задаваемых вопросов по STL:
<ftp://butler.hpl.hp.com/stl/stl.faq>
- Безопасная реализация STL Кея Хорстмана:
<http://www.horstmann.com/safestl.html>
- Список ресурсов STL Уоррена Янга:
<http://www.cyberport.com/~tangent/programming/stl/resources.html>

Библиография

Ниже перечислены книги и другие информационные ресурсы, которые упоминались, использовались или цитировались в этой книге, а также список книг, позволяющих получить дополнительную информацию по рассматриваемой теме. Очевидно, что список не является исчерпывающим. Здесь перечислены только те источники, которые считает полезными автор.

1. Matthew H. Austern
Generic Programming and the STL.
Using and Extending the C++ Standard Template Library
Addison-Wesley, Reading, MA, 1998
2. Ulrich Breymann
Komponenten entwerfen mit der STL
Addison-Wesley, Bonn, Germany, 1999
3. Bernd Eggink
Die C++ iostreams-Library
Hanser Verlag, München, Germany, 1995
4. Margaret A. Ellis, Bjarne Stroustrup
The Annotated C++ Reference Manual (ARM)
Addison-Wesley, Reading, MA, 1990
5. Graham Glass, Brett Schuchert
The STL <Primer>
Prentice-Hall, Englewood Cliffs, NJ, 1996
6. ISO
Information Technology — Programming Languages — C++
Document Number ISO/IEC 14882-1998
ISO/IEC, 1998
7. Scott Meyers
More Effective C++
35 New Ways to Improve Your Programs and Designs
Addison-Wesley, Reading, MA, 1996

8. David R. Musser, Atul Saini
STL Tutorial and Reference Guide
C++ Programming with the Standard Template Library
Addison-Wesley, Reading, MA, 1996
9. Mark Nelson
C++ Programmer's Guide to the Standard Template Library
IDG Books Worldwide, Foster City, CA, 1995
10. ObjectSpace
Systems <Toolkit> UNIX Reference Manual
ObjectSpace, 1995
11. P. J. Plauger
The Draft Standard C++ Library
Prentice Hall, Englewood Cliffs, NJ, 1995
12. Bjarne Stroustrup
The C++ Programming Language, 3rd edition
Addison-Wesley, Reading, MA, 1997
13. Bjarne Stroustrup
The Design and Evolution of C++
Addison-Wesley, Reading, MA, 1994
14. Steve Teale
C++ IOStreams Handbook
Addison-Wesley, Reading, MA, 1993

Алфавитный указатель

A

abort(), 85
abs(), 131, 522, 551, 557
accumulate(), 414
acos(), 551, 556
address(), 712
adjacent_difference(), 419
adjacent_find(), 351
adjustfield, 593
advance(), 265
allocate(), 702, 712
allocator_type, 235, 253, 508
alnum, 690
alpha, 690
always_noconv(), 695
any(), 448
app, 606
append(), 473, 497
apply(), 549
arg(), 522
argc, 37, 608
argv, 37, 608
asin(), 551, 556
assign(), 160, 171, 176, 241, 472, 660
at(), 161, 172, 242, 495
atan(), 551, 556
atan2(), 551, 556
ate, 606
atexit(), 85
auto_ptr, 54, 56, 67
auto_ptr_ref, 69

B

back(), 161, 243, 435
back_inserter, 118, 277

bad(), 573
bad_alloc, 43
bad_cast, 43
bad_exception, 44
bad_typeid, 44
badbit, 571
base(), 274
basefield, 596
basic_filebuf, 602
basic_fstream, 602
basic_ifstream, 602
basic_ios, 563
basic_istream, 563
basic_iostream, 619
basic_ofstream, 602
basic_ostream, 563
basic_ostringstream, 620
basic_streambuf, 641
basic_string, 462
basic_stringbuf, 620
basic_stringstream, 620
beg, 610
begin(), 96, 162, 177, 190, 207, 244
binary, 606
binary_function, 312
binary_search(), 400
bind1st, 307
bind2nd, 142, 307
bitset, 444
Blitz, 525
bool
 ввод, 592
 вывод, 592
 тип, 34

boolalpha, 592

Boost, 229

C

C, локальный контекст, 667

c_str, 495, 605

capacity(), 157, 238, 492

catalog, 699

ceil(), 556

cerr, 560, 613

char, 73, 689

char_traits, 565, 660

char_type, 660, 661

cin, 560

classic(), 675

classic_table, 691

clear(), 93, 165, 171, 250, 500, 573

clog, 560

close(), 699

cntrl, 690

codecvt, 693

collate, 697

combine(), 674

compare(), 493, 660, 698

complex, 510

compose_f_gx, 313

compose_f_gx_hx, 313

compose_f_gx_hy, 313

compose_f_gxy, 313

conj(), 521

const_cast, 36

const_iterator, 98, 234

const_mem_fun_ref_t, 310

const_mem_fun_t, 310

const_mem_fun1_ref_t, 310

const_mem_fun1_t, 310

const_pointer, 490, 712

const_reference, 234, 489, 711

const_reverse_iterator, 235

construct(), 702, 712

container_type, 434

copy(), 358

copy_backward(), 358

copy_event, 635

copyfmt(), 591

copyfmt(), 634

cos(), 525, 551, 556

cosh(), 525, 551, 556

count(), 337, 448

count_if(), 337

cout, 560, 613

ctype_base, 690

cur, 610

C-строки, 454

D

data(), 495

date_order, 682

dateorder, 683

deallocate(), 702, 712

dec

манипулятор, 597

флаг, 596

decimal_point, 685

denorm_absent, 77

denorm_indefinite, 77

denorm_min(), 74

denorm_present, 77

destroy(), 702, 712

difference_type, 235, 489, 710

digit, 690

digits, 74, 446

digits10, 74

distance(), 265, 267, 290

div(), 557

divides, 305

domain_error, 45

double, 73

dynamic_cast, 35

E

element_type, 66

empty(), 176, 187, 204, 237, 434

end, 610

end(), 96, 162, 190, 207, 507

endl, 561

ends, 561

eof(), 660
 eofbit, 571
 epptr(), 642
 epsilon(), 74
 eq(), 660
 eq_int_type(), 660
 equal(), 352
 equal_range(), 188, 404
 equal_to(), 305
 erase(), 500
 erase_event, 635
 error, 695
 event_callback, 634
 exit(), 85
 EXIT_FAILURE, 85
 EXIT_SUCCESS, 85
 exp(), 524, 551, 556
 explicit, 34
 export, 26

F

fabs(), 556
 fail(), 573
 failbit, 571
 failed(), 639
 false, 34
 falsename(), 678
 filebuf, 602
 fill(), 366
 find(), 340
 find_end(), 347
 find_first_of(), 349
 find_if(), 340
 first, 52
 first_argument_type, 311
 first_type, 51
 fixed, 598
 flags(), 591
 flip(), 167, 449
 float, 73
 floor(), 556
 flush, 561
 flush(), 585
 for_each(), 333

fpos, 609
 frac_digits, 685
 freeze(), 624
 front(), 161, 243, 434
 front_inserter(), 278
 fstream, 602

G

gcount(), 584
 generate(), 368
 generate_n(), 368
 get(), 68, 688, 699
 get_allocator(), 253, 702
 get_date(), 682
 get_monthname(), 682
 get_temporary_buffer(), 705
 get_time(), 682
 get_weekday(), 682
 get_year(), 682
 getline(), 582
 getloc(), 637
 global(), 675
 good(), 573
 goodbit, 571
 greater, 305
 greater_equal, 305
 grouping(), 685
 gslice_array, 526, 553

H

has_denorm, 74
 has_denorm_loss, 74
 has_INFINITY, 74
 has_quiet_NaN, 74
 has_signaling_NaN, 74
 hex
 манипулятор, 597
 флаг, 596

I

imbue_event, 635
 in, 606
 includes(), 401

indirect_array, 526, 555
inner_product(), 416
inplace_merge(), 413
ios_base, 45
IOSStream, 558
is_iec559, 74
is_integer, 74
is_specialized, 74
iterator, 234, 490
iword(), 634

K

key_compare, 235
key_type, 235

L

labs(), 557
LANG, 664
ldexp(), 556
ldiv(), 557
length(), 468
length_error, 45
less, 305
less_equal, 305
lexicographical_compare(), 356
log(), 524, 551
log10(), 524, 551
logic_error, 44
logical_and, 305
logical_not, 305
logical_or, 305
lower_bound(), 402
l-значения, 70

M

main(), 37
make_heap(), 397
mapped_type, 235
mask_array, 526, 554
max(), 74
max_element(), 338
max_exponent, 74
max_exponent10, 74

mem_fun, 307
mem_fun_ref, 307
merge(), 406
min(), 74
min_element(), 338
mismatch(), 354
modf(), 556
modulus, 305

N

negate, 305
new, 43
next_permutation(), 383
none(), 448
norm(), 522
not1, 307
not2, 307
npos, 490
nth_element(), 394
NULL, 84

O

oct
манипулятор, 597
флаг, 596
offsetof(), 84
out, 606
out_of_range, 45
overflow_error, 45

P

pair, 50, 51
partial_sort(), 391
partial_sort_copy(), 393
partial_sum(), 418
partition(), 387
pbase(), 642
plus, 305
pointer, 489
polar(), 520
pop_heap(), 397
pow(), 524, 551, 556
ptr(), 642

prev_permutation(), 383
 priority_queue, 438
 ptr_fun, 310
 pubimbue(), 637
 pubseekoff(), 637
 pubseekpos(), 637
 push_heap(), 397
 pword(), 634

Q

queue, 430
 quiet_NaN, 74

R

radix, 74
 rand(), 557
 random_shuffle(), 385
 range_error, 45
 rbegin(), 162, 190, 207, 507
 rdstate(), 573
 reference, 234, 489
 reinterpret_cast, 36
 release(), 69
 remove(), 371
 remove_copy(), 373
 remove_copy_if(), 373
 remove_if(), 371
 rend(), 162, 190, 207, 507
 replace(), 369
 replace_copy(), 370, 394
 replace_copy_if(), 370
 replace_if(), 369
 reset(), 449
 result_type, 311
 return_temporary_buffer(), 705
 reverse(), 379
 reverse_copy(), 379
 reverse_iterator, 234, 490
 rotate(), 380
 rotate_copy(), 381
 round_error(), 74
 round_to_nearest, 77
 round_toward_infinity, 77

round_toward_neg_infinity, 77
 round_toward_zero, 77
 runtime_error, 45

S

search(), 345
 search_n(), 342
 second, 52
 second_argument_type, 311
 second_type, 51
 set(), 449
 set_difference(), 409
 set_intersection(), 408
 set_symmetric_difference(), 410
 set_union (), 407
 setstate(), 573
 sin(), 525, 556
 sinh(), 525, 556
 size_type, 434
 size(), 448, 468
 size_t, 84
 size_type, 235, 489
 slice_array, 526, 551
 sort(), 389
 sort_heap(), 397
 sqrt(), 524, 556
 srand(), 557
 stable_partition(), 387
 stable_sort(), 389
 static_cast, 35
 swap_ranges(), 365

T

tan(), 525
 tanh(), 525
 test(), 448
 throw, 32
 tinyness_before, 74
 traits_type, 489
 transform(), 362
 traps, 74
 true, 34
 trunc, 606

typeid, 44
typename, 27

U

underflow_error, 45
unexpected(), 44
Unicode, 463
unique(), 375
unique_copy(), 377
upper_bound(), 402
using
 директива, 33
 объявление, 33

V

valarray, 525
value_compare, 235
value_type, 234, 433, 489, 520

W

wfilebuf, 602
wfstream, 602
what(), 46
ws, 561

A

адаптеры
 контейнерные, 422
 функциональные, 306
алгоритм
 модифицирующий, 324, 358
 немодифицирующий, 323, 336
 перестановочный, 326, 378
 сортировки, 327, 388
 удаления, 326, 371
 упорядоченных интервалов,
 330, 399
 численный, 331, 414
алгоритмы
 accumulate(), 414
 adjacent_difference(), 419
 adjacent_find(), 351

алгоритмы (*продолжение*)
 binary_search(), 400
 copy(), 358
 copy_backward(), 358
 count_if(), 337
 coutn(), 337
 equal(), 352
 equal_range(), 404
 fill(), 366
 find(), 340
 find_end(), 347
 find_first_of(), 349
 find_if(), 340
 for_each(), 333
 generate(), 368
 generate_n(), 368
 includes(), 401
 inner_product(), 416
 inplace_merge(), 413
 lexicographical_compare(), 356
 lower_bound(), 402
 make_heap(), 397
 max_element(), 338
 merge(), 406
 min_element(), 338
 mismatch(), 354
 next_permutation(), 383
 partial_sort(), 391
 partial_sort_copy(), 393
 partial_sum(), 418
 partition(), 387
 pop_heap(), 397
 prev_permutation(), 383
 push_heap(), 397
 random_shuffle(), 385
 remove(), 371
 remove_copy(), 373
 remove_copy_if(), 373
 remove_if(), 371
 replace(), 369
 replace_copy(), 370
 replace_copy_if(), 370
 replace_if(), 369

алгоритмы (*продолжение*)

reverse(), 379
reverse_copy(), 379
rotate(), 380
rotate_copy(), 381
search(), 345
search_n(), 342
set_difference(), 409
set_intersection(), 408
set_symmetric_difference(), 410
set_union(), 407
sort(), 389
sort_heap(), 397
stable_partition(), 387
stable_sort(), 389
swap_ranges(), 365
transform(), 362
unique(), 375
unique_copy(), 377
upper_bound(), 402

- амортизированная сложность, 38
- асимметричность, 184
- ассоциативные контейнеры, 89, 94, 99, 280

ассоциативные массивы, 95, 212

6

базовые гарантии, 149
бинарный предикат, 133
битовые поля, 444

B

векторы, 156
возможности, 157
емкость, 157
заголовочный файл, 156
немодифицирующие
операции, 159
обращение к элементам, 161
присваивание, 160
размер, 157
вещественная часть, 510
вложенные классы, 30

1

- двоичное представление, 445
- дву направленные итераторы, 257
- деки, 169
 - возможности, 170
 - заголовочный файл, 169
 - конструкторы, 171
 - обработка исключений, 173
 - операции, 171
 - присваивание, 171
- деструкторы
 - auto_ptr, 68
 - векторы, 159
 - контейнеры, 236
 - локальные контексты, 672
 - множества, 185
 - мультимножества, 185
 - мультиотображения, 203
 - отображения, 203
 - распределители, 712
 - списки, 176
 - строки, 491

E

емкость, 157, 468

3

заголовочные файлы. 320

1

- иерархия классов
 - исключения, 43
 - классы строковых потоков, 620
 - потоковые классы, 562
 - файловые потоки, 602
- интернационализация, 657
- Интернет, 22
- иррефлексивность, 184
- исключение
 - иерархия, 43
 - класс, 43
 - обявление, 32
 - спецификация, 32

исключения, 46
bad_alloc, 43
bad_cast, 43
bad_exception, 44
bad_typeid, 44
exception, 43

итераторы
ввода, 257
вывода, 257
пользовательские, 291
произвольного доступа, 257
теги, 287
трактовка, 287, 288

К

квадратичная сложность, 38
квазиупорядоченность, строгая, 184
коллекции, 86, 356
комплексные числа, 510
конечный итератор вставки, 277
константы
 EXIT_FAILURE, 85
 EXIT_SUCCESS, 85
 NULL, 84

конструкторы
 complex, 520
 битовые поля, 447
 векторы, 159
 деки, 171
 контейнеры, 236
 локальные контексты, 667
 массивы значений, 527
 множества, 185
 мультимножества, 185
 мультиотображения, 203
 отображения, 203
 очереди, 434
 пары, 51
 распределители, 711
 списки, 176
 стеки, 422, 426
 строки, 490
 шаблонные, 29

контейнеры, 88, 144, 422
адаптеры, 422
векторы, 90
деки, 91
инициализация, 139, 154
итераторы, 96
конструкторы, 236
обработка исключений, 254
подсчет ссылок, 226
пользовательские, 224
списки, 92
сравнение, 229
ссылочная семантика, 144
строки, 94
требования к элементам, 143
узловые, 149
копирующий конструктор, 29
куча, 396

Л

линейная сложность, 38
логарифмическая сложность, 38
локальные контексты, 664

М

массивы
 ассоциативные, 95, 212
 значений, 525
 операции, 527
 проблемы, 532
 трансцендентные функции, 530
 как контейнеры STL, 224
 контейнеры, 162
мнимальная часть, 510
многобайтовая кодировка, 463
множества, 183
 возможности, 184
 заголовочный файл, 183
 обработка исключений, 194
 операции, 185
мультимножества, 183
 возможности, 184
 заголовочный файл, 183

мультимножества (*продолжение*)

обработка исключений, 194

операции, 185

мультиотображения, 200, 201

Н

начальный итератор вставки, 117, 278

О

обработка исключений, 42

обратные итераторы, 270

обратный вызов, 634

общий итератор вставки, 279

объекты функций, 134, 295

оператор преобразования типа, 35

операторы

`const_cast`, 36

`dynamic_cast`, 35

`reinterpret_cast`, 36

`static_cast`, 35

`typeid`, 44

отложенное вычисление, 532

отображения, 200, 201

очереди, 430

заголовочный файл, 430

основной интерфейс, 431

П

последовательные контейнеры, 88

постоянная сложность, 38

потоковые итераторы, 281

правило Кенига, 33

приоритетные очереди, 438

присваивание

векторы, 156

деки, 171

итераторы, 96

справки, 176

пространства имен, 32

`std`, 40

правило Кенига, 33

прямые итераторы, 257

Р

раскрутка стека, 31

распределители, 49, 701

С

символы

классификация, 689

преобразование кодировки, 693

трактовки, 660

словари, 95

сложность, 37

справки, 174

возможности, 175

заголовочный файл, 174

обращение к элементам, 177

операции, 176

присваивание, 176

сравнения

интервальные, 356

лексикографические, 356

срезы, 533

стеки, 422

заголовочный файл, 423

основной интерфейс, 423

строгая квазиупорядоченность, 184

Т

транзакционная безопасность, 149

транзитивность, 184

У

указатели

`auto_ptr`, 54

`NULL`, 84

умные указатели, 54, 145

унарный предикат, 131

Ф

файловые дескрипторы, 611

файлы, 603, 610

фацеты, 670, 676

`messages`, 699

`money_get`, 688

`money_put`, 687

фацеты (*продолжение*)

moneyrunc, 684

num_get(), 680

num_put, 679

numrunc, 678

time_get, 682

time_put, 683

форматирование

bool, 592

вещественные числа, 598

форматированный ввод-вывод, 589

форматные флаги, 589

функции, 134, 295

функции

как аргументы, 95

как критерии сортировки, 133

Ш

шаблоны, 25

турепаме, 27

вложенные классы, 30

конструкторы, 29

копирующий конструктор, 52

нетипизованные параметры, 26

параметры по умолчанию, 27

функции, 28

Н. Джосьюотис

C++ Стандартная библиотека. Для профессионалов
Перевел с английского Е. Матвеев

Главный редактор	<i>E. Строганова</i>
Заведующий редакцией	<i>И. Корнеев</i>
Руководитель проекта	<i>А. Чижова</i>
Научный редактор	<i>Е. Матвеев</i>
Литературный редактор	<i>А. Жданов</i>
Художник	<i>Н. Биржаков</i>
Корректор	<i>Н. Викторова</i>
Верстка	<i>Р. Гришианов</i>

Лицензия ИД № 05784 от 07.09.2001.

Подписано в печать 20.10.03. Формат 70×100/16. Усл. п. л. 59,34.

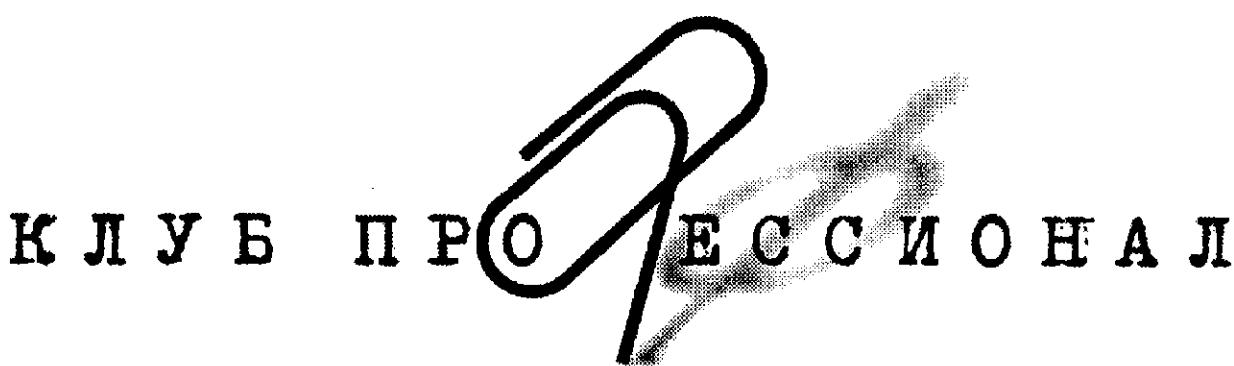
Тираж 3000 экз. Заказ № 868.

ООО «Питер Принт». 196105, Санкт-Петербург, ул. Благодатная, д. 67в.

Налоговая льгота — общероссийский классификатор продукции

OK 005-93, том 2; 953005 — литература учебная.

Отпечатано с готовых диапозитов в ФГУП «Печатный двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.



В 1997 году по инициативе генерального директора Издательского дома «Питер» Валерия Степанова и при поддержке деловых кругов города в Санкт-Петербурге был основан «Книжный клуб Профессионал». Он собрал под флагом клуба профессионалов своего дела, которых объединяет постоянная тяга к знаниям и любовь к книгам. Членами клуба являются лучшие студенты и известные практики из разных сфер деятельности, которые хотят стать или уже стали профессионалами в той или иной области.

Как и все развивающиеся проекты, с течением времени книжный клуб вырос в «Клуб Профессионал». Идею клуба сегодня формируют три основные «клубные» функции:

- неформальное общение и совместный досуг интересных людей;
- участие в подготовке специалистов высокого класса (семинары, пакеты книг по специальной литературе);
- формирование и высказывание мнений современного профессионала (при встречах и на страницах журнала).

КАК ВСТУПИТЬ В КЛУБ?

Для вступления в «Клуб Профессионал» вам необходимо:

- ознакомиться с правилами вступления в «Клуб Профессионал» на страницах журнала или на сайте www.piter.com;
- выразить свое желание вступить в «Клуб Профессионал» по электронной почте postbook@piter.com или по тел. (812) 103-73-74;
- заказать книги на сумму не менее 500 рублей в течение любого времени или приобрести комплект «Библиотека професионала».

«БИБЛИОТЕКА ПРОФЕССИОНАЛА»

Мы предлагаем вам получить все необходимые знания, подписавшись на «Библиотеку професионала». Она для тех, кто экономит не только время, но и деньги. Покупая комплект – книжную полку «Библиотека професионала», вы получаете:

- скидку 15% от розничной цены издания, без учета почтовых расходов;
- при покупке двух или более комплектов – дополнительную скидку 3%;
- членство в «Клубе Профессионал»;
- подарок – журнал «Клуб Профессионал».

Закажите бесплатный журнал
«Клуб Профессионал».

издательский дом
ПИТЕР[®]
www.PITER.COM

КНИГА-ПОЧТОЙ



**ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:**

- по телефону: (812) 103-73-74;
- по электронному адресу: postbook@piter.com;
- на нашем сервере: www.piter.com;
- по почте: 197198, Санкт-Петербург, а/я 619
ЗАО «Питер Пост».

**ВЫ МОЖЕТЕ ВЫБРАТЬ ОДИН ИЗ ДВУХ СПОСОБОВ ДОСТАВКИ
И ОПЛАТЫ ИЗДАНИЙ:**

- (✉) Наложенным платежом с оплатой заказа при получении посылки на ближайшем почтовом отделении. Цены на издания приведены ориентировочно и включают в себя стоимость пересылки по почте (**но без учета авиатарифа**). Книги будут высланы нашей службой «Книга-почтой» в течение двух недель после получения заказа или выхода книги из печати.
- (✉) Оплата наличными при курьерской доставке (**для жителей Москвы и Санкт-Петербурга**). Курьер доставит заказ по указанному адресу в удобное для вас время в течение трех дней.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, код, количество заказываемых экземпляров.

Вы можете заказать бесплатный журнал «Клуб Профессионал».

издательский дом
ПИТЕР[®]
WWW.PITER.COM



Нет времени ходить по магазинам?

наберите:

www.piter.com

Здесь вы найдете:

Все книги издательства сразу

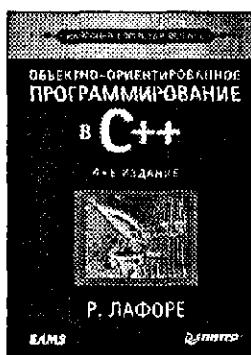
Новые книги — в момент выхода из типографии

Информацию о книге — отзывы, рецензии, отрывки

Старые книги — в библиотеке и на CD

**И, наконец, вы нигде не купите
наши книги дешевле!**

КНИГА-ПОЧТОЙ



928 с., 17×24,
перепл.
Код 7804

Цена наложенным
платежом 497 р.

R. Lafore

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В С++. КЛАССИКА COMPUTER SCIENCE

Благодаря этой книге тысячи пользователей овладели технологией объектно-ориентированного программирования в С++. В ней есть все: основные принципы языка, готовые полномасштабные приложения, небольшие примеры, поясняющие теорию, и множество полезных иллюстраций.

Книга пользуется стабильным успехом в учебных заведениях благодаря тому, что содержит более 100 упражнений, позволяющих проверить знания по всем темам.

Читатель может вообще не иметь подготовки в области языка С++. Необходимо лишь знание начальных основ программирования.



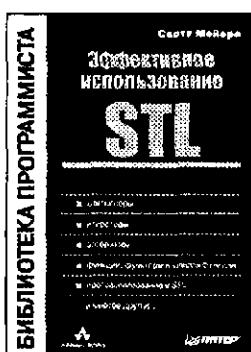
464 с., 17×24, перепл.
Код 2112
Цена наложенным
платежом 206 р.

Т. Павловская

С/С++. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ: УЧЕБНИК ДЛЯ ВУЗОВ

Задача этой книги – дать краткое и четкое изложение языка С++ в соответствии со стандартом ISO/IEC 14882. Учебник предназначен в первую очередь для студентов, изучающих язык «с нуля», но и более искушенные в программировании специалисты найдут в нем немало полезной информации. В книге рассматриваются принципы объектно-ориентированного программирования и их реализация на С++, средства, возможности и конструкции языка, приводятся практические примеры, дается толчок к дальнейшему изучению этого и других языков программирования.

Контрольные задания по ключевым темам представлены в 20 вариантах, и автор надеется, что преподаватели достойно оценят проявленную о них заботу.



224 с., 17×24, обл.
Код 1133
Цена наложенным
платежом 154 р.

С. Мейерс

ЭФФЕКТИВНОЕ ИСПОЛЬЗОВАНИЕ STL. БИБЛИОТЕКА ПРОГРАММИСТА

В этой книге известный автор Скотт Мейерс раскрывает секреты настоящих мастеров, позволяющие добиться максимальной эффективности при работе с библиотекой STL.

Во многих книгах описываются возможности STL, но только в этой рассказано о том, как работать с этой библиотекой. Каждый из 50 советов книги подкреплен анализом и убедительными примерами, поэтому читатель узнает не только, как решать ту или иную задачу, но и когда следует выбирать то или иное решение – и почему именно такое.



СПЕЦИАЛИСТАМ
КНИЖНОГО БИЗНЕСА!

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают эксклюзивный ассортимент компьютерной, медицинской,
психологической, экономической и популярной литературы

РОССИЯ

Москва м. «Калужская», ул. Бутлерова, д. 17б, офис 207, 240; тел./факс (095) 777-54-67;
e-mail: sales@piter.msk.ru

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а;
тел. (812) 103-73-73, факс (812) 103-73-83; e-mail: sales@piter.com

Воронеж ул. 25 января, д. 4; тел. (0732) 27-18-86;
e-mail: piter-vrn@mail.ru; piterv@comch.ru

Екатеринбург ул. 8 Марта, д. 267б; тел./факс (3432) 25-39-94; e-mail: piter-ural@r66.ru

Нижний Новгород ул. Премудрова, д. 31а; тел. (8312) 58-50-15, 58-50-25;
e-mail: piter@infonet.nnov.ru

Новосибирск ул. Немировича-Данченко, д. 104, офис 502;
тел/факс (3832) 54-13-09, (3832) 47-92-93; e-mail: piter-sib@risp.ru

Ростов-на-Дону ул. Калитвинская, д. 17в; тел. (8632) 95-36-31, (8632) 95-36-32;
e-mail: jupiter@rost.ru

Самара ул. Новосадовая, д. 4; тел. (8462) 37-06-07; e-mail: piter-volga@sama.ru

УКРАИНА

Харьков ул. Сузdalские ряды, д. 12, офис 10–11, т. (057) 712-27-05;
e-mail: piter@tender.kharkov.ua

Киев пр. Красных Казаков, д. 6, корп. 1; тел./факс (044) 490-35-68, 490-35-69;
e-mail: office@piter-press.kiev.ua

БЕЛАРУСЬ

Минск ул. Бобруйская д., 21, офис 3; тел./факс (37517) 226-19-53; e-mail: piter@mail.by

МОЛДОВА

Кишинев «Ауратип-Питер»; ул. Митрополит Варлаам, 65, офис 345; тел. (3732) 22-69-52,
факс (3732) 27-24-82; e-mail: lili@auratip.mldnet.com



Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 103-73-73.**

E-mail: grigorjan@piter.com



Издательский дом «Питер» приглашает к сотрудничеству авторов.
Обращайтесь по телефонам: Санкт-Петербург — **(812) 327-13-11,**
Москва — (095) 777-54-67.



Заказ книг для вузов и библиотек: **(812) 103-73-73.**
Специальное предложение — e-mail: kozin@piter.com



УВАЖАЕМЫЕ ГОСПОДА!
КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
ВЫ МОЖЕТЕ ПРИОБРЕСТИ
ОПТОМ И В РОЗНИЦУ
У НАШИХ РЕГИОНАЛЬНЫХ ПАРТНЕРОВ.

Башкортостан

Уфа, «Азия», ул. Зенцова, д. 70 (оптовая продажа),
маг. «Оазис», ул. Чернышевского, д. 88,
тел./факс (3472) 50-39-00.
E-mail: asiaufa@ufanet.ru

Дальний Восток

Владивосток, «Приморский торговый дом книги»,
тел./факс (4232) 23-82-12.
E-mail: bookbase@mail.primorye.ru

Хабаровск, «Мирс»,
тел. (4212) 30-54-47, факс 22-73-30.
E-mail: sale_book@bookmirs.khv.ru

Хабаровск, «Книжный мир»,
тел. (4212) 32-85-51, факс 32-82-50.
E-mail: postmaster@worldbooks.kht.ru

Европейские регионы России

Архангельск, «Дом книги»,
тел. (8182) 65-41-34, факс 65-41-34.
E-mail: book@atnet.ru

Калининград, «Вестер»,
тел./факс (0112) 21-56-28, 21-62-07.
E-mail: nshibkova@vester.ru
<http://www.vester.ru>

Северный Кавказ

Ессентуки, «Россы», ул. Октябрьская, 424,
тел./факс (87934) 6-93-09.
E-mail: rossy@kmw.ru

Сибирь

Иркутск, «Продалитъ»,
тел. (3952) 59-13-70, факс 51-30-70.
E-mail: prodalit@irk.ru
<http://www.prodalit.irk.ru>

Иркутск, «Антей-книга»,
тел./факс (3952) 33-42-47.
E-mail: antey@irk.ru

Красноярск, «Книжный мир»,
тел./факс (3912) 27-39-71.
E-mail: book-world@public.krasnet.ru

Нижневартовск, «Дом книги»,
тел. (3466) 23-27-14, факс 23-59-50.
E-mail: book@nvartovsk.wsnet.ru

Новосибирск, «Топ-книга»,
тел. (3832) 36-10-26, факс 36-10-27.
E-mail: office@top-kniga.ru
<http://www.top-kniga.ru>

Тюмень, «Друг»,
тел./факс (3452) 21-34-82.
E-mail: drug@tyumen.ru

Тюмень, «Фолиант»,
тел. (3452) 27-36-06, факс 27-36-11.
E-mail: foliant@tyumen.ru

Челябинск, ТД «Эврика», ул. Барбюса, д. 61,
тел./факс (3512) 52-49-23.
E-mail: evrika@chel.surnet.ru

Татарстан

Казань, «Таис»,
тел. (8432) 72-34-55, факс 72-27-82.
E-mail: tais@bancorp.ru

Урал

Екатеринбург, магазин № 14,
ул. Челюскинцев, д. 23,
тел./факс (3432) 53-24-90.
E-mail: gvardia@mail.ur.ru

Екатеринбург, «Валео-книга»,
ул. Ключевская, д. 5,
тел./факс (3432) 42-56-00.
E-mail: valeo@etel.ru