STL для программистов на C++

Леен Аммераааль

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

STL for C++ Programmers

Leen Ammeraal
Hogeschool Utrecht, The Netherlands

JOHN WILEY & SONS
Chichester • New York • Weinhwim • Brisbane • Toronto • Sungapore

STL для программистов на C++

Леен Аммерааль
Высшая школа Утрехта, Нидерланды

ДМК
издательство
Москва

Леен Аммерааль
STL для программистов на C++. Пер. с англ./Леен Аммерааль – М.: ДМК. – 240 с., ил.

ISBN 5-89818-027-3

Книга Леена Аммерааля посвящена стандартной библиотеке шаблонов (STL) – мощному инструменту повышения эффективности труда программистов, пишущих на C++.

Умелое использование STL позволяет повысить надежность, переносимость и универсальность программ, а также снизить расходы на их разработку. В книге описана стандартизованная версия STL. Дается введение в предмет, которое позволяет быстро освоить библиотеку шаблонов. Приведен исчерпывающий справочный материал, в том числе по новому классу STL, string. Изложение сопровождается многочисленными примерами небольших, но законченных программ, иллюстрирующих ключевые понятия STL. Особое внимание уделено разъяснению сложных понятий библиотеки шаблонов, например, функциональных объектов и адаптеров функций.

Книга предназначена как для профессиональных программистов и тех, кто углубленно изучает C++, так и для тех, кто только начинает осваивать этот язык программирования, без преувеличения самый популярный в мире.

ISBN 0-471-97181-2

ISBN 5-89818-027-3

Содержание

Предисловие

Когда несколько лет назад в языке C++ появились шаблоны, лишь немногие из программистов на C++ могли предположить, какое влияние это окажет на стандарт библиотеки языка. Стандартная библиотека шаблонов (Standard Template Library) была первоначально разработана сотрудниками Hewlett-Packard А.А. Степановым и М. Ли совместно с Д.Р. Муссером из Ренсселэровского политехнического института. После внесения незначительных поправок Комитет по стандартизации языка C++ принял STL, сделав ее существенной составной частью стандартной библиотеки.

Использование STL дает возможность создавать более надежные, более переносимые и более универсальные программы, а также сократить расходы на их разработку. Это значит, что ни один профессиональный программист не может себе позволить пройти мимо этой библиотеки. Я написал книгу для таких программистов, а также для людей, в достаточной мере знакомых с C++.

В книге описана стандартная версия STL, а не изначальный вариант, разработанный в Hewlett-Packard. Вы можете загрузить примеры, приводимые в этой книге, из сети Internet не только для того, чтобы сэкономить на набивании кода, но и для упрощения проблем с переносимостью: некоторые из электронных версий примеров сделаны более переносимыми путем добавления условной компиляции, чтобы обойти нестандартное поведение версий STL, поставляемых с компиляторами Borland и Microsoft. Все примеры доступны в виде одного файла, *stlcpp.zip*, на моем Web-сайте по адресу:

http://www.econ.hvu.nl/~ameraal/

или напрямую с одного из следующих ftp-сайтов:

ftp://ftp.expa.fnt.hvu.nl/pub/ameraal

ftp://pitel-lnx.ibk.fnt.hvu.nl/pub/ameraal

Я благодарен Гэйнору Редверс-Маттону из издательства Wiley и Фрэнсису Глассброу из Ассоциации пользователей C и C++, которые убедили меня написать эту книгу и дали полезные рекомендации по ее содержанию.

1

STL для начинающих

1.1. Шаблоны, пространства имен и тип *bool*

Как легко догадаться из названия, стандартная библиотека шаблонов (STL) основывается на относительно новом понятии *шаблона*. Поэтому мы начнем с краткого обсуждения этого предмета.

Шаблонные функции

Предположим, что для некоторого положительного числа *x* нам приходится часто вычислять значение выражения

```
2 * x + (x * x + 1) / (2 * x)
```

где *x* может быть типа *double* или *int*. В последнем случае оператор деления / обозначает целочисленное деление, дающее целый результат. Например, если *x* имеет тип *double* и равен 5.0, тогда значение приведенного выражения составляет 12.6, но если *x* имеет тип *int* и равен 5, то значение выражения будет 12. Вместо того чтобы писать две функции, такие как

```
double f(double x)
{  double x2 = 2 * x;
   return x2 + (x * x + 1)/x2;
}
```

```
int f(int x)
{  int x2 = 2 * x;
   return x2 + (x * x + 1)/x2;
}
```

нам достаточно создать один *шаблон*, как показано в следующем примере, который представляет собой законченную программу:

```
// ftempl.cpp: Шаблонная функция.
#include <iostream.h>

template <class T>
T f(T x)
{  T x2 = 2 * x;
   return x2 + (x * x + 1)/x2;
}

int main()
{  cout << f(5.0) << endl << f(5) << endl;
   return 0;
}
```

Программа выведет

```
12.6
12
```

В этом шаблоне *T* – тип, задаваемый аргументом при вызове *f*. При вызове *f*(5.0) T будет обозначать *double* (это тип константы 5.0), так что, к примеру, в выражении (*x* * *x* + 1)/*x2* выполнится деление с плавающей точкой. Напротив, при исполнении вызова *f* (5) T будет обозначать тип *int*, что приведет к целочисленному делению.

При разборе программы *ftempl.cpp* компилятор создает две различные функции, весьма похожие на функции *f*(*double*) и *f*(*int*), с которых мы начинали наш пример. Следовательно, компилятор должен одновременно «видеть» как определения, так и вызовы шаблонов. Это делает шаблоны плохими кандидатами на раздельную компиляцию; вместо этого мы, как правило, помещаем шаблоны в файлы заголовка. Когда мы используем файлы заголовка, написанные кем-то другим, мы не видим определения шаблонов и вызываем их как обычные функции, что показано на примере вызовов *f*(5.0) и *f*(5) в нашей программе. Поэтому, применяя шаблоны функций STL, мы можем и не знать, что вызываем функции, созданные из шаблонов.

Что в имени?

Шаблон, который, подобно рассмотренному выше, начинается со слова *template*, а заканчивается закрывающей фигурной скобкой, следующей

за оператором *return*, создателем языка Бьерном Страуструпом был изначально назван *шаблоном функции*. Данный термин отражает, что мы имеем дело с определенным видом шаблона, отличающимся от шаблонов классов, о которых речь пойдет ниже. Сегодня многие авторы используют вместо этого выражение *шаблонная функция (template funcion)*, потому что шаблоны указанного типа очень похожи на обычные функции. В этой книге мы также будем использовать термин *шаблонная функция*, а иногда даже просто *функция* для обозначения этих шаблонов. То же относится и к обсуждаемому ниже понятию, которое изначально получило наименование *шаблон класса*, а в книге зовется *шаблонным классом (template class)* или просто *классом*.

Шаблонные классы

Мы можем использовать тип как параметр (*T* в предыдущем примере) для классов почти так же, как и для функций. Предположим, нам нужен класс *Pair*, чтобы хранить пары значений. Иногда оба значения принадлежат к типу *double*, иногда к типу *int*. Тогда вместо двух новых классов, к примеру,

```
class PairDouble {
public:
   PairDouble(double x1, double y1): x(x1), y(y1) {}
   void showQ();
private:
   double x, y;
};

void PairDouble::showQ()
{  cout << x/y << endl;
}
```

после чего следует аналогичный фрагмент с классом *PairInt*, нам достаточно написать один *шаблонный класс*:

```
// cltempl.cpp: Шаблонный класс.
#include <iostream.h>

template <class T>
class Pair {
public:
   Pair(T x1, T y1): x(x1), y(y1){}
   void showQ();
private:
   T x, y;
};

template <class T>
```

```
void Pair<T>::showQ()
{  cout << x/y << endl;
}

int main()
{  Pair<double> a(37.0, 5.0);
   Pair<int> u(37, 5);
   a.showQ();
   u.showQ();
   return 0;
}
```

Способ, каким функции-члены шаблонного класса, в этом примере *showQ*, определены вне класса, может с первого взгляда показаться сложным. Но на самом деле эта конструкция весьма логична: имя любой функции-члена, когда эта функция определяется вне класса, должно быть предварено выражением

```
type::
```

и вполне резонно, что вместо *type* в нашем случае мы пишем *Pair<T>*. Кроме того, как пользователи STL мы можем не беспокоиться об определениях, так как шаблонные классы STL доступны в виде файлов заголовков, которые можно использовать, не вдаваясь в подробности их программирования. Единственный аспект применения шаблонов, который мы увидим в наших программах,– это обозначение фактического типа с помощью конструкции наподобие *Pair<double>*.

Пространства имен

Существует другой новый элемент языка, который мы обязаны принять во внимание. Если программа состоит из многих файлов, мы должны принять меры во избежание *конфликта имен*. Концепция пространства имен может быть хорошим способом решения этой задачи. В нижеприведенной программе определены две глобальные переменные *i*, которые не находятся в конфликте, потому что существуют в различных пространствах имен:

```
// namespac.cpp: Концепция пространства имен.
#include <iostream.h>

namespace A
{ int i = 10;
}

namespace B
{ int i = 20;
}
```

```
void fA()
{  using namespace A;
   cout << "In fA:   " <<
      A::i << " " << B::i << " " << i << endl;
}

void fB()
{  using namespace B;
   cout << "In fB:   " <<
      A::i << " " << B::i << " " << i << endl;
}

int main()
{  fA(); fB();
   cout << "In main: " << A::i << " " << B::i << endl;
   // cout << i << endl; Здесь это недопустимо.
   using A::i;
   cout << i << endl; // А это разрешено.
   return 0;
}
```

Эта программа на выходе даст:

```
In fA:   10 20 10
In fB:   10 20 20
In main: 10 20
10
```

Благодаря идентификаторам *A* и *B* мы впоследствии можем ссылаться на эти пространства имен. Для пространства имен *A* можем написать либо что-нибудь вроде

```
A:: ...
```

либо одно из выражений:

```
using namespace A;
using A::i;
```

Только после использования одного из двух последних выражений неквалифицированный идентификатор *i* будет относиться к переменной *i* (со значением 10), определенной в пространстве имен *A*. Результат работы программы наглядно демонстрирует это.

Тип *bool*: синоним для *int* или встроенный тип?

Тип *bool* и два его возможных значения, *true* и *false*, определены в файлах заголовка первоначальной версии STL с помощью приема, который часто можно встретить в программах на C:

```
#define bool int
#define true 1
#define false 0
```

Однако в соответствии с проектом стандарта C++ *bool* является встроенным типом, что подразумевает: следующая программа, вообще не использующая файлы заголовков, должна компилироваться без ошибок:

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

Некоторые старые компиляторы отвергнут эту программу, поскольку они не распознают *bool* в качестве встроенного типа.

В соответствии с проектом стандарта C++ типы *bool* и *int* не являются идентичными, что проиллюстрируем следующими (типичными) выходными результатами программы *boolint.cpp*:

```
sizeof(bool) = 1
sizeof(int) = 4
With B defined as bool B[100], we have
sizeof(B) = 100
```

Программа, которая выводит такие результаты, приведена ниже:

```
// boolint.cpp: Типы bool и int различны.
#include <iostream.h>

int main()
{  cout << "sizeof(bool) = " << sizeof(bool) << endl;
   cout << "sizeof(int) = " << sizeof(int) << endl;
   bool B[100];
   cout << "With B defined as bool B[100], we have\n";
   cout << "sizeof(B) = " << sizeof(B) << endl;
   return 0;
}
```

Очевидно, каждый элемент массива значений типа *bool* занимает один байт, тогда как при размере машинного слова в 32 бита он занимал бы 4 байта, если бы типы *bool* и *int* не различались. Можно представить еще более экономную реализацию, когда восемь булевских значений размещаются в одном байте, но это замедлило бы операции с этими элементами.

В следующем разделе мы обсудим дополнительные различия между версиями C++ (а также STL).

1.2. Знакомство с STL

Установив современный компилятор C++, мы можем сразу начать использовать STL, например откомпилировать и запустить следующую программу. Эта программа читает с клавиатуры переменное количество ненулевых целых чисел и печатает их в том же порядке после того, как введен 0. Данная задача может показаться слишком простой, но на самом деле это не так, потому что нет ограничения на количество вводимых чисел:

```
// readwr.cpp: Чтение и вывод переменного количества
//               ненулевых целых (ввод завершается нулем).
#include <iostream>
#include <vector>
using namespace std;

int main()
{  vector<int> v;
   int x;
   cout << "Enter positive integers, followed by 0:\n";
   while (cin >> x, x != 0)
      v.push_back(x);
   vector<int>::iterator i;
   for (i=v.begin(); i != v.end(); ++i)
      cout << *i << " ";
   cout << endl;
   return 0;
}
```

Мы можем использовать шаблон *vector* как массив переменной длины. Сначала эта длина равна нулю. Поскольку мы хотим, чтобы элементы вектора были целого типа, то всегда пишем *vector<int>*, чтобы обозначить класс, с которым работаем. Выражение

```
v.push_back(x);
```

добавляет значение *x* типа *int* в конец вектора *v*.

Оператор *for* в этой программе используется аналогично тому, как это сделано в следующем фрагменте кода, который выводит массив *a* вместо вектора *v*:

```
int a[N], *p;
...
for (p=a; p != a+N; p++)
   cout << *p << " ";
```

Напомним, что выражения *&a*[0] и *a* эквивалентны, равно как и выражения *&a*[*N*] и *a* + *N*. Начав с первого элемента, мы проходим массив, пока

не оказываемся за его концом: хотя указываем на адрес *a*[*N*], последний элемент массива – *a*[*N*–1]. Это может выглядеть опасным, но поскольку мы не используем значение *a*[*N*], а только его адрес, такой стиль абсолютно безопасен. Переменная *i*, определенная как

```
vector<int>::iterator i;
```

называется *итератором*. Она используется таким же образом, как указатель в вышеприведенном фрагменте. Значение итератора для первого элемента вектора *v* обозначается выражением *v.begin*(), а значение итератора для элемента, следующего за конечным, обозначается как *v.end*(). Значение элемента вектора, на который ссылается допустимый итератор *i*, обозначается выражением * *i*, как если бы *i* был указателем. Для этого итератора *i* определены также операторы ++ и ––, как в префиксном, так и в суффиксном варианте. Это объясняет смысл следующего цикла *for*:

```
for (i=v.begin(); i != v.end(); ++i)
      cout << *i << " ";
```

В приведенном цикле лучше *не* заменять != на <. Хотя в примере это сработает, но оператор < неприменим к некоторым другим типам, отличными от *vector<int>* (см. раздел 1.9), в то время как оператор != работает во всех случаях.

Обычно в математике запись [*a, b*] используется для обозначения закрытого интервала $a \leqslant x \leqslant b$, а запись (*a, b*) – для открытого интервала $a < x < b$. Это объясняет запись

```
[a, b)
```

для интервала

$$a \leqslant x < b$$

Подобным же образом мы иногда будем писать

```
[ia, ib)
```

для диапазона значений итератора в следующем фрагменте кода:

```
vector<int>::iterator i, ia, ib;
...
for (i = ia; i != ib; ++i) ...
```

Ошибка выделения памяти

Поскольку все числа, которые читает программа *readwr.cpp*, хранятся в динамически распределяемой памяти, используемая нами компьютерная система наложит ограничение на размер ввода. Вопрос усложняется из-за

реализации некоторыми операционными системами виртуальной памяти, что предполагает использование жесткого диска для расширения оперативной памяти. Хотя этот подход предоставляет в наше распоряжение огромное количество памяти (за счет уменьшения скорости вычислений), ясно, что рано или поздно фрагмент кода

```
vector<int> v;
for (;;) v.push_back(0);
```

приведет к ошибке выделения памяти. То же самое наблюдается при исполнении

```
int *p;
for (;;) p = new int;
```

В последнем фрагменте обычная проверка *if* (*p* != *NULL*) не будет работать с современными компиляторами C++. Согласно проекту стандарта C++ ошибка выделения памяти будет приводить не к возвращению значения *NULL,* а к «выбросу исключения». Несмотря на то что ошибка распределения памяти относится к C++, а не к STL, это была бы достаточно важная тема для обсуждения в настоящей книге, если бы существовало простое, переносимое решение, совместимое с большинством популярных компиляторов и соответствующее принятому стандарту C++. Поскольку различные компиляторы требуют разных подходов, а стандарт языка находится в стадии проекта, мы опустим обсуждение этой темы в данной книге, которая все-таки посвящена STL, а не C++.

Возвращаясь назад

Рекомендация использовать != (или ==) для сравнения значений итераторов вместо <, <=, >, >= усложняет прохождение элементов вектора в обратном порядке с помощью только что обсужденных средств. В программе *readwr.cpp* мы могли бы добиться этого, заменив цикл *for* на следующий фрагмент:

```
i = v.end();
if (i != v.begin())
   do cout << *--i << " "; while (i != v.begin());
```

В начале приведенного решения требуется дополнительное сравнение для определения пустого вектора, в случае если число 0 было единственным числом, введенным пользователем программы. Однако существует более простой путь прохождения вектора (и других структур данных) задом наперед. Он требует использования двух других функций-членов, *rbegin* и *rend,* вместе с другим типом итератора, *reverse_iterator*, как демонстрирует следующий фрагмент:

```
vector<int>::reverse_iterator i;
for (i=v.rbegin; i != v.rend(); ++i)
   cout << * i << " ";
```

Заметьте, что в этом случае мы пишем ++*i* вместо --*i*.

Новые элементы языка и проблемы переносимости

В программе *readwr.cpp* способ указания файлов заголовка в строках *#include* может показаться вам непривычным (и отличающимся от того, как мы писали в разделе 1.1). Раньше мы всегда использовали *iostream.h* и *vector.h* вместо просто *iostream* и *vector*. Эти короткие формы приняты в последней версии C++, которая называется проектом стандарта C++. Во многих случаях можно использовать любой из вариантов, например, указывая как <*iostream.h*>, так и <*iostream*>. Далее мы везде будем пользоваться последним, более новым вариантом.

Интересно, что, хотя мы пишем <*iostream*>, на самом деле файл может называться *iostream.h* (например, для компилятора Borland C++ 5.2). В связи с этим возникают сомнения, правомерно ли называть *iostream* файлом заголовка. С этого раздела начнем использовать термин *заголовок*, а не *файл заголовка*. В соответствии с принятым употреблением мы будем применять этот короткий термин не только к именам, заключенным в угловые скобки, как <*iostream*>, но и к самим файлам, таким как *iostream.h*.

Другой новый аспект языка заставит нас добавлять конструкцию *using namespace std* в начале программы. Если опустим эту строчку, может появиться сообщение об ошибке типа «*Не определен символ 'vector'*». Преимущества использования концепции пространства имен, как мы обсуждали в разделе 1.1, заключаются в том, что имена наподобие *vector* не «загрязняют глобальное пространство имен». Иными словами, мы вправе использовать слово *vector* на глобальном уровне для любой другой цели. Если мы поступим так, то при необходимости разрешить неоднозначность напишем ::*vector* для глобальной версии и *std*::*vector* – для версии STL.

Два популярных компилятора C++

Как Visual C++ 5.0 (VC5), так и Borland C++ 5.2 (BC5) поддерживают пространства имен. STL полностью интегрирована в их библиотеки, так что не требуется никаких специальных действий, чтобы начать ее использование. Многие используют эти компиляторы из интегрированной среды разработки, с помощью которой возможно редактировать программные файлы, компилировать их и т. д. Вместо этого мы будем компилировать и компоновать программы из командной строки с помощью команды *cl* или *bcc32*.

Для этого в командную строку PATH-файла *autoexec.bat* должен быть вставлен путь

```
c:\progra~1\devstudio\vc\bin
```

или

```
c:\progra~1\borland\cbuilder\bin
```

Для работы с VC5 необходимо также ввести команду

```
vcvars32
```

чтобы указать компилятору, где следует искать включаемые файлы и библиотеки.

Версии STL

Изначальная версия STL от компании Hewlett-Packard составила основу значительной части проекта стандарта C++. Эта версия доступна в сети Internet и может быть бесплатно использована и модифицирована при соблюдении нестрогих условий, указанных в следующем уведомлении о копирайте:
Версия STL, включенная в проект стандарта C++, отличается от изначальной (которую мы будем называть HP STL) количеством и именами заголовков. Вместо 48 файлов в HP STL в проекте стандарта C++ используется только 13 (см. табл.).

```
Copyright (c) 1994, Hewlett-Packard
Company.
Разрешение использовать, делать копии,
модифицировать, распространять
и продавать это программное обеспечение
и сопутствующую документацию в любых целях
предоставляется настоящим уведомлением
при условии, что как вышеприведенное
уведомление о копирайте, так и настоящее
уведомление об использовании будет
приведено в сопутствующей документации.
Компания Hewlett-Packard не несет
ответственности за пригодность
к использованию этого программного
обеспечения в каких бы то ни было целях.
Это программное обеспечение
предоставляется «как есть», без явных либо
неявных гарантий.
```

Заголовки HP STL		Стандартная STL	
algo.h	algobase.h	bool.h	algorithm
bvector.h	defalloc.h	deque.h	deque
faralloc.h	fdeque.h	flist.h	functional
fmap.h	fmultmap.h	fmultset.h	iterator
fset.h	function.h	hdeque.h	list
heap.h	hlist.h	hmap.h	map
hmultmap.h	hmultset.h	hset.h	memory
hugalloc.h	hvector.h	iterator.h	numeric
lbvector.h	ldeque.h	list.h	queue
llist.h	lmap.h	lmultmap.h	set
lmultset.h	lngalloc.h	lset.h	stack
map.h	multimap.h	multiset.h	utility
neralloc.h	nmap.h	nmultmap.h	vector
nmultset.h	nset.h	pair.h	
projectn.h	set.h	stack.h	
tempbuf.h	tree.h	vector.h	

Как VC5, так и BC5 соответствуют проекту стандарта C++ по названиям заголовков. Если у вас более старый компилятор (вместе с HP STL), вероятно, он потребует, чтобы имена заголовков содержали в себе *.h*, так что вам придется писать *<vector.h>* и *<iostream.h>* вместо *<vector>* и *<iostream>*. В этом случае будет также необходимо убрать операторы *using*, вроде того, что предшествует функции *main* в программе *readwr.cpp*.

Стандартная библиотека шаблонов фирмы SGI, адаптированная для VC++ 5.0

Хотя VC5 очень хороший компилятор, в версии STL, которая с ним поставляется, имеются некоторые проблемы. Поскольку они могут быть разрешены в следующем релизе этого компилятора, не будем их подробно обсуждать. В то же время пользователи Visual C++ могут выбрать версию STL, написанную проектировщиком STL Александром Степановым для фирмы Silicon Graphics (SGI) и адаптированную для VC5 Уэйном Учида (Wayne Ouchida). Следующая Web-страничка содержит подробную информацию по этому вопросу:

```
http://www.sirius.com/~ouchida/
```

Версия SGI STL часто упоминается в связи с ее высоким качеством, и эта адаптация поможет программистам, работающим с Visual C++, воспользоваться этим преимуществом.

В тех случаях, где современные версии STL отличаются друг от друга, за основу для примеров в этой книге берется проект стандарта C++ (на декабрь 1996 года). Если у вас возникнут проблемы с переносимостью примеров для компиляторов Microsoft или Borland, попробуйте использовать электронную версию этих программ, доступную в виде файла *stdcpp.zip*. Некоторые из них сделаны более переносимыми с помощью условной компиляции.

Кроме того, существуют также коммерческие версии STL, которые мы не будем здесь обсуждать. Поскольку STL состоит только из файлов заголовка, замена одной версии на другую является легко решаемой задачей.

1.3. Векторы, списки и двусторонние очереди

В программе *readwr.cpp* три раза встречается слово *vector*:

```
#include <vector>
...
vector<int> v;
...
vector<int>::iterator i;
```

Применение концепции вектора обеспечивает выделение непрерывной памяти, для чего программисты на C обычно пользуются функциями *malloc*, *realloc* и *free*. В качестве альтернативы можно употребить связный список, как рекомендуется в книгах по структурам данных. С помощью STL мы можем использовать (двойные) связные списки, не программируя их самостоятельно. Все, что нам требуется для программы *readwr.cpp*,– заменить всюду слово *vector* на *list*, как показано в следующей программе:

```
// readwr1.cpp: Чтение и вывод переменного количества
//              ненулевых целых (ввод завершается нулем).
//              Использует список.
#include <iostream>
#include <list>
using namespace std;

int main()
{  list<int> v;
   int x;
   cout << "Enter positive integers, followed by 0:\n";
   while (cin >> x, x != 0)
      v.push_back(x);
   list<int>::iterator i;
   for (i=v.begin(); i != v.end(); ++i)
      cout << *i << " ";
   cout << endl;
   return 0;
}
```

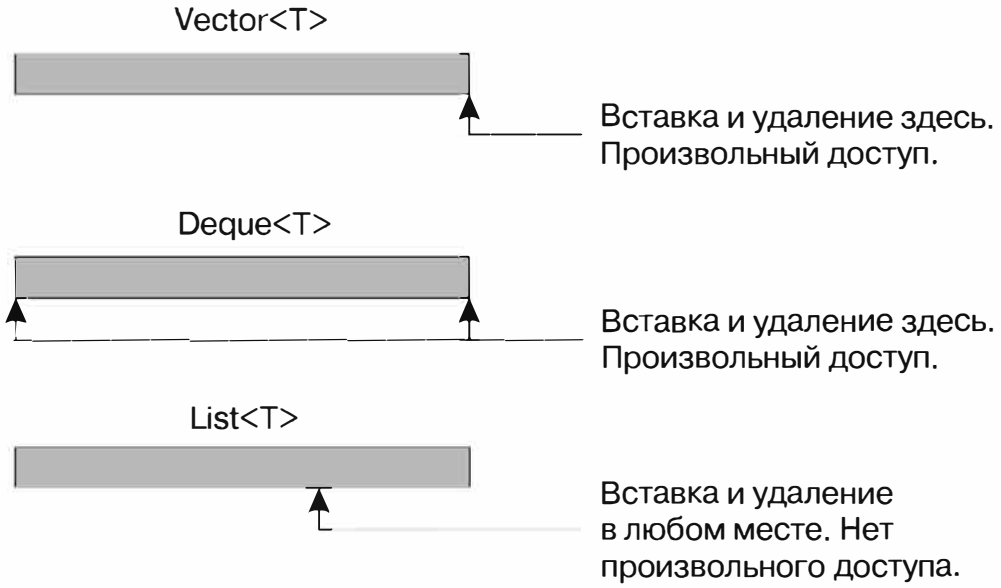

Рисунок 1.1. Свойства трех последовательных контейнеров

Программа также будет правильно выполняться, если заменить слово *list* на *deque* (двусторонняя очередь), что дает нам третье решение. Пользователь не заметит никаких различий в поведении этих трех версий программы, но внутреннее представление данных будет различаться. Это скажется на наборе доступных операций, которые смогут выполняться эффективно.

Для данного типа *T* типы *vector<T>*, *deque<T>* и *list<T>* называются *последовательными контейнерами*.

Как показано на рисунке 1.1, мы можем эффективно вставлять и удалять элементы только в конце вектора, в начале и конце двусторонней очереди и в любом месте списка. Хотя вектор накладывает сильные ограничения на вставку и удаление, он имеет преимущество в том, что предоставляет произвольный доступ к своим элементам. Некоторые дальнейшие примеры прояснят эти различия. Для полноты изложения необходимо отметить, что существует четвертая разновидность последовательного контейнера, обычный массив, который описывается как

```
T a[N];
```

для массива с элементами типа *T*. Далее продемонстрируем, что средства STL предоставляют возможность успешно отсортировать массив. Другими словами, STL может быть полезна даже в программах, которые используют обычные массивы, а не типичные контейнеры STL, о чем рассказано в следующем разделе.

Как показано в приведенной ниже таблице, векторы, двусторонние очереди и списки поддерживают различные наборы операций.

Операция	Функция	vector	deque	list
Вставить в конце	*push_back*	✓	✓	✓
Удалить в конце	*pop_back*	✓	✓	✓
Вставить в начале	*push_front*	-	✓	✓
Удалить в начале	*pop_front*	-	✓	✓
Вставить в любом месте	*insert*	(✓)	(✓)	✓
Удалить в любом месте	*erase*	(✓)	(✓)	✓
Отсортировать (см. раздел 1.4)	*sort* (алгоритм)	✓	✓	-

Галочка в скобках (✓) означает, что функции *insert* и *erase*, хотя определены для векторов и двусторонних очередей, применительно к этим контейнерам выполняются гораздо медленнее, чем для списков. Говорят, что их выполнение занимает *линейное время* для векторов и двусторонних очередей, а это означает: время их выполнения пропорционально длине последовательности, хранящейся в контейнере. В противоположность этому все операции, помеченные галочкой ✓ (без скобок), выполняются за *постоянное время*, то есть время, необходимое для их выполнения, не зависит от длины последовательности.

До сих пор мы видели только, как используется функция *push_back*. Следующая программа показывает, как использовать все функции для вставки и удаления, перечисленные в вышеприведенной таблице (*push_back*, *pop_back*, *push_front*, *pop_front*, *insert* и *erase*).

```
// insdel.cpp: Вставка и удаление элементов из списка.
#include <iostream>
#include <list>
using namespace std;

void showlist(const char *str, const list<int> &L)
{  list<int>::const_iterator i;
   cout << str << endl << "  ";
   for (i=L.begin(); i != L.end(); ++i)
      cout << *i << " ";
   cout << endl;
}

int main()
{  list<int> L;
   int x;
   cout << "Enter positive integers, followed by 0:\n";
   while (cin >> x, x != 0)
      L.push_back(x);
   showlist("Initial list:", L);
   L.push_front(123);
```

```
    showlist("After inserting 123 at the beginning:", L);
    list<int>::iterator i = L.begin();
    L.insert(++i, 456);
    showlist(
       "After inserting 456 at the second position:", L);
    i = L.end();
    L.insert(--i, 999);
    showlist(
       "After inserting 999 just before the end:", L);
    i = L.begin(); x = *i;
    L.pop_front();
    cout << "Deleted at the beginning: " << x << endl;
    showlist("After this deletion:", L);
    i = L.end(); x = *--i;
    L.pop_back();
    cout << "Deleted at the end: " << x << endl;
    showlist("After this deletion:", L);
    i = L.begin();
    x = *++i; cout << "To be deleted: " << x << endl;
    L.erase(i);
    showlist("After this deletion (of second element):",
       L);
    return 0;
}
```

Функции для вставки и удаления здесь применяются к списку, поскольку это единственный контейнерный тип, для которого все эти функции определены и выполняются эффективно, как указывает вышеприведенная таблица. В следующем примере выполнения программы (и далее в этой книге) данные, вводимые с клавиатуры, подчеркнуты:

```
Enter positive integers, followed by 0:
10 20 30  0
Initial list:
  10 20 30
After inserting 123 at the beginning:
  123 10 20 30
After inserting 456 at the second position:
  123 456 10 20 30
After inserting 999 just before the end:
  123 456 10 20 999 30
Deleted at the beginning: 123
After this deletion:
  456 10 20 999 30
Deleted at the end: 30
After this deletion:
  456 10 20 999
```

```
To be deleted: 10
After this deletion (of second element):
  456 20 999
```

Употребление *const* в первых двух строчках функции *showlist* требует разъяснения:

```
void showlist(const char *str, const list<int> &L)
{  list<int>::const_iterator i;
```

Добавление приставки *const* к параметрам типа указатель или ссылка, как это сделано выше, является хорошей практикой, если такие параметры не используются для модификации объектов, на которые они указывают. Поскольку функция *showlist* не модифицирует ни строку *str*, ни список *L*, отсюда происходят два употребления слова *const* на первой из этих двух строчек. Из-за этого на второй строчке мы должны объявить переменную *i* типа *const_iterator*, чтобы иметь возможность использовать ее вместе с *L*. Это похоже на применение модификатора *const* к указателям: если хотим присвоить вышеобъявленный параметр *str* указателю *p*, мы сможем сделать это, только использовав *const* при объявлении этого указателя:

```
const char *p;    // const необходим, поскольку str
p = str;          // типа const char *
```

Стирание подпоследовательности

Если [*i*1, *i*2) является действительным диапазоном для вектора *v*, мы можем стереть подпоследовательность в *v*, заданную этим диапазоном, следующим образом:

```
v.erase(i1, i2);
```

То же самое относится и к остальным контейнерам.

1.4. Сортировка

Ниже следует расширение программы *readwr.cpp*, рассмотренной в начале раздела 1.2. Эта программа сортирует вектор *v*, то есть располагает элементы *v* в восходящем порядке:

```
// sort1.cpp: Сортировка вектора.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{  vector<int> v;
```

```
    int x;
    cout << "Enter positive integers, followed by 0:\n";
    while (cin >> x, x != 0) v.push_back(x);
    sort(v.begin(), v.end());
    cout << "After sorting: \n";
    vector<int>::iterator i;
    for (i=v.begin(); i != v.end(); ++i)
       cout << *i << " ";
    cout << endl;
    return 0;
}
```

Сама сортировка выполняется выражением

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

Мы используем два значения итераторов в качестве аргументов: первый, *v.begin*(), ссылается на начальный элемент вектора и второй, *v.end*(), ссылается на следующий за последним элемент вектора. Вывод этой программы содержит введенные пользователем целые числа, отсортированные в восходящем порядке.

Вышеприведенный вызов функции *sort* отличается от вызовов *push_back*, *insert*, *begin* и др. Поскольку мы пишем не *v.sort*(...), а просто *sort*(...), видно, что *sort* является не функцией-членом класса *vector*, а *шаблонной функцией*, которая не является членом класса. Технический термин, обозначающий такую шаблонную функцию в STL,– *обобщенный* (*generic*) *алгоритм*, или просто *алгоритм*. Строчка

```
#include <algorithm>
```

необходима, поскольку мы используем алгоритм *sort*. Для некоторых реализаций STL компилятор не выдает сообщения об ошибках, если мы опустим эту строчку, поскольку заголовок *algorithm* включается неявным образом с помощью строки

```
#include <vector>
```

Так как было бы неразумно рассчитывать на это, мы используем обе строчки *#include*. Напротив, в следующей программе, которая сортирует обычный массив вместо вектора, достаточно включить только заголовок *algorithm*:

```
// sort2.cpp: Сортировка массива.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
```

```
{  int a[10], x, n = 0, *p;
   cout << "Enter at most 10 positive integers, "
      "followed by 0:\n";
   while (cin >> x, x != 0 && n < 10) a[n++] = x;
   sort(a, a+n);
   cout << "After sorting: \n";
   for (p=a; p != a+n; p++) cout << *p << " ";
   cout << endl;
   return 0;
}
```

Важно заметить сходство между вызовами

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

в программе *sort1.cpp* и

```
sort(a, a+n);
```

в программе *sort2.cpp*. В обоих случаях первый аргумент ссылается на первый элемент последовательного контейнера, а второй – на элемент, следующий за последним, точнее на позицию, находящуюся непосредственно после последнего элемента. Это общий принцип, справедливый не только для алгоритма *sort*, но и для большинства алгоритмов STL. Мы можем применить этот принцип также и к заданной подпоследовательности. Например, мы можем отсортировать только элементы $a[3]$, $a[4]$, $a[5]$ и $a[6]$, написав

```
sort(a+3, a+7);
```

или, что эквивалентно,

```
sort[&a[3], &a[7]);
```

как показывает рисунок 1.2.

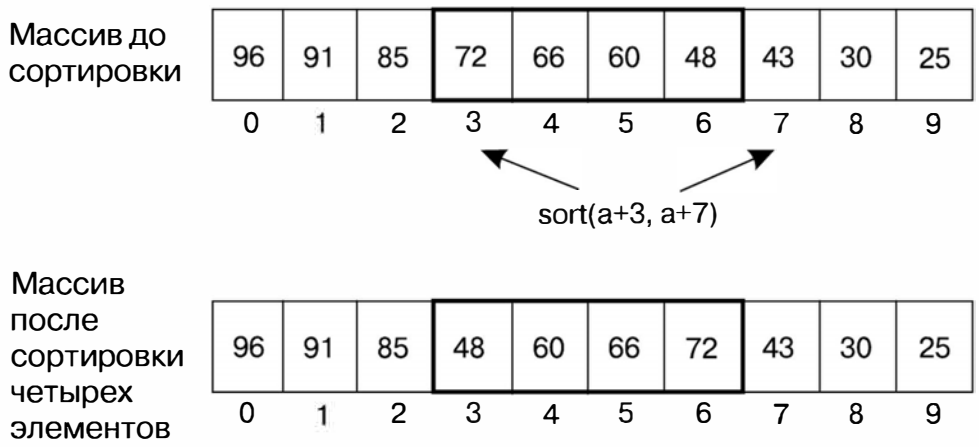

Рисунок 1.2. Сортировка подпоследовательности

Может показаться более логичным передавать адрес последних элементов, а не следующих за ними, в этом примере *a*[6] вместо *a*[7]. Однако использование элементов, следующих за последним, имеет несколько преимуществ. Например, мы можем определить количество элементов с помощью простого вычитания:

количество элементов = 7 – 4 = 3

Принятое соглашение также позволяет нам написать цикл *for*:

```
for (i=3; i!=7; ++i) ...
```

если мы хотим сделать что-либо с отсортированными элементами. Возможность выбрать подпоследовательность применима и к контейнерам STL, таким как векторы. Например, если в программе *sort1.cpp* вектор *v* также содержит не менее семи элементов, отсортируем *v*[3], *v*[4], *v*[5] и *v*[6], написав

```
vector<int>::iterator i, j;
i = v.begin() + 3;
j = v.begin() + 7;
sort(i, j);
```

или просто

```
sort(v.begin() + 3, v.begin() + 7);
```

Такое использование оператора + рассмотрим более подробно в разделе 1.9.

Произвольный доступ, доступ по индексу и сортировка

Вы могли заметить, что в предыдущем обсуждении мы использовали выражения типа *v*[3], хотя *v* не является массивом, а определен как

```
vector<int> v;
```

Доступ по индексу возможен в этом случае, поскольку вектор является контейнером произвольного доступа, для которого определен *operator*[] доступа по индексу. Заметьте, однако, что мы не можем заменить *v*[3] на *(*v* + 3), потому что тип переменной *v* является классом, для которого не определен ни бинарный оператор +, ни унарный оператор *.

Алгоритм STL *sort* требует произвольного доступа. Поскольку такой доступ обеспечивают векторы и массивы, мы могли использовать этот алгоритм в программах *sort1.cpp* и *sort2.cpp*. Как показывает рисунок 1.1, двусторонняя очередь также обеспечивает произвольный доступ, в отличие от списка. Это объясняет, почему вызов

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

встречающийся в программе *sort1.cpp,* будет работать, если мы в программе заменим всюду *vector* на *deque*, но не на *list*. Более подробно – см. раздел 1.9.

Инициализация контейнеров

Как известно каждому программисту на C(++), определение массива может содержать список начальных значений. Например, мы можем написать

```
int a[3] = {10, 5, 7};
int b[] = {8, 13};
   // эквивалентно int b[2] = {8, 13};
int c[3] = {4};
   // эквивалентно int c[3] = {4, 0, 0};
```

Инициализация также возможна и для трех других типов последовательных контейнеров, как показано в следующем примере:

```
int a[3] = {10, 5, 7};
vector<int> v(a, a+3);
deque<int> w(a, a+3);
list<int> x(a, a+3);
```

Но не только массив, а также и вектор, двусторонняя очередь или список могут служить основой для инициализации контейнера того же типа. Например, если мы продолжим этот пример строчкой

```
vector<int> v1(v.begin(), v.end());
```

вектор *v*1 станет идентичен вектору *v*, оба будут состоять из трех элементов типа *int*, 10, 5 и 7. Этот пример откомпилируется, поскольку *v* и *v*1 относятся к одному типу – *vector<int>*. Напротив, следующий пример не будет компилироваться, поскольку мы не можем использовать значения списка *x* для инициализации вектора *v*1[1]:

```
vector<int> v1(x.begin(), x.end());
```

Ясно, что этот способ инициализации обеспечивается *конструкторами* контейнерных классов. Кроме этого, существуют конструкторы, принимающие в качестве параметра целое, означающее желаемый размер, и необязательный параметр, задающий значение для элементов, содер-

[1] В окончательном стандарте C++ нет этого ограничения, поэтому приведенный пример *должен* откомпилироваться без ошибок на более позднем компиляторе, но рассматриваемые автором компиляторы VC5 и BC5 основываются на раннем варианте стандарта языка, который не поддерживает версии STL, позволяющие инициализировать контейнер одного типа элементами контейнера другого типа (за исключением массива). – *Прим. переводчика.*

жащихся во вновь созданном контейнере. Например, мы можем написать

```
vector<int> v(5, 8) // Пять элементов, все они равны 8.
```

или

```
vector<int> v(5);
```

В последнем случае вектор *v* будет содержать пять элементов. Эта форма записи удобна в том случае, когда мы хотим присвоить значения элементам позже, как в примере из раздела 1.6.

1.5. Алгоритм *find*

Следующая программа показывает, как мы можем находить требуемое значение в векторе:

```
// find1.cpp: Найти заданное значение в векторе.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{  vector<int> v;
   int x;
   cout << "Enter positive integers, followed by 0:\n";
   while (cin >> x, x != 0)
      v.push_back(x);

   cout << "Value to be searched for: ";
   cin >> x;
   vector<int>::iterator i =
      find(v.begin(), v.end(), x);
   if (i == v.end())
      cout << "Not found\n";
   else
   {  cout << "Found";
      if (i == v.begin())
         cout << " as the first element";
      else cout << " after " << *--i;
   }
   cout << endl;
   return 0;
}
```

Алгоритм *find* применим к каждому из четырех последовательных контейнеров (вектор, двусторонняя очередь, список и массив). Если мы везде заменим *vector* на *deque*, поведение программы не изменится, как и в случае, если мы заменим *vector* на *list*. Версия для массива целочисленных значений приведена ниже:

```
// find2.cpp: Найти заданное значение в массиве.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{  int a[10], x, n = 0;
   cout << "Enter at most 10 positive integers, "
      "followed by 0:\n";
   while (cin >> x, x != 0 && n < 10) a[n++] = x;
   cout << "Value to be searched for: ";
   cin >> x;
   int *p = find(a, a+n, x);
   if (p == a+n)
      cout << "Not found\n";
   else
   {  cout << "Found";
      if (p == a)
         cout << " as the first element";
      else cout << " after " << *--p;
   }
   cout << endl;
   return 0;
}
```

1.6. Алгоритм *copy* и итератор вставки

Мы можем использовать алгоритм *copy* для копирования элементов одного контейнера в другой, причем, например, источником может быть вектор, а приемником – список, как показывает следующая программа:

```
// copy1.cpp: Копируем вектор в список.
//            Первая версия: режим замещения.
#include <iostream>
#include <vector>
#include <list>
using namespace std;

int main()
{  int a[4] = {10, 20, 30, 40};
   vector<int> v(a, a+4);
   list<int> L(4);          // Список из 4 элементов
   copy(v.begin(), v.end(), L.begin());
   list<int>::iterator i;
   for (i=L.begin(); i != L.end(); ++i)
      cout << *i << " ";  // Результат: 10 20 30 40
   cout << endl;
   return 0;
}
```

Режимы замещения и вставки

Поскольку в список *L* необходимо скопировать четыре элемента, при его определении ему задана длина 4:

```
list<int> L(4);  // Список из 4 элементов.
```

Это необходимо, поскольку алгоритм *copy,* когда его используют таким способом, работает в *режиме замещения*. Это напоминает то, как вводятся символы с клавиатуры при наборе текста в режиме замещения. В этом режиме мы пишем поверх имеющегося текста; многие, однако, предпочитают *режим вставки*. На обычной клавиатуре клавиша *Ins* переключает режим вставки в режим замещения и обратно. Что касается алгоритма *copy*, то для него также не составит труда перейти в режим вставки. Попробуем начать с пустого списка, заменив вышеприведенное определение *L* на следующее:

```
list<int> L;    // Пустой список.
```

Затем заменим вызов алгоритма *copy* на следующий:

```
  copy(v.begin(), v.end(), inserter(L, L.begin()));
```

С этими двумя изменениями программы содержимое списка *L* станет таким же, как и в изначальной версии. Новая запись

```
inserter(...)
```

зовется *итератором вставки*, который является особым видом *адаптера итератора*, о чем рассказано в разделе 6.6. Мы можем также использовать его, чтобы вставить данные в середину последовательности, как показывает следующая версия:

```
// copy2.cpp: Копируем вектор в список.
//            Вторая версия: режим вставки.

#include <iostream>
#include <vector>
#include <list>
using namespace std;

int main()
{  int a[4] = {10, 20, 30, 40};
   vector<int> v(a, a+4);
   list<int> L(5, 123);  // Список из 5 элементов
   list<int>::iterator i = L.begin();
   ++i; ++i;
   copy(v.begin(), v.end(), inserter(L, i));
```

```
    for (i=L.begin(); i != L.end(); ++i)
       cout << *i << " ";
    cout << endl;
    return 0;
}
```

Начальное содержимое списка *L* таково:

```
123 123 123 123 123
```

Затем итератор *i* устанавливается таким образом, чтобы он указывал на третий элемент списка, после чего он используется в выражении

```
inserter(L, i)
```

которое вычисляется при вызове алгоритма *copy*. В результате третий элемент списка и оба элемента, следующие за ним, сдвигаются на четыре позиции вправо, а на место третьего элемента вставляются значения 10, 20, 30 и 40. Это объясняет следующий результат работы данной программы:

```
123 123 10 20 30 40 123 123 123
```

Ввод и вывод

Интересно, что мы можем использовать алгоритм *copy* для ввода и вывода, как будет показано в разделе 1.9.

1.7. Алгоритм *merge*

Рисунок 1.3 иллюстрирует операцию объединения:
Алгоритм объединения *merge* может быть использован для каждого из четырех типов последовательных контейнеров (массивы, векторы, двусторонние очереди и списки). Удивительно, что три участника алгоритма (*a*, *b* и *c* на рисунке 1.3) не обязаны принадлежать к одному и тому же контейнерному типу. Чтобы продемонстрировать этот факт, давайте объединим вектор *a* и массив *b* в список *c*:

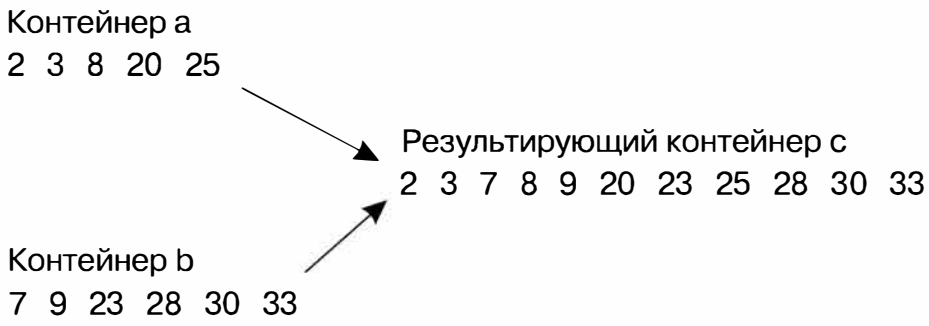

Рисунок 1.3. Объединение **a** и **b** в **c**

```
// merge.cpp: Объединение вектора и массива в список.
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>

using namespace std;

int main()
{  vector<int> a(5);
   a[0] = 2; a[1] = 3; a[2] = 8;
   a[3] = 20; a[4] = 25;
   int b[6] = {7, 9, 23, 28, 30, 33};
   list<int> c; // Список с сначала пуст
   merge(a.begin(), a.end(), b, b+6,
      inserter(c, c.begin()));
   list<int>::iterator i;
   for (i=c.begin(); i != c.end(); ++i)
      cout << *i << " ";
   cout << endl;
   return 0;
}
```

Как и в случае с *copy*, нам приходится использовать итератор вставки, если мы хотим писать в список *c* в режиме вставки. В качестве альтернативы мы могли бы написать

```
list<int> c(11); // принимаем 5 + 6 = 11 элементов
merge(a.begin(), a.end(), b, b+6, c.begin());
```

выделив достаточно места при определении принимающего списка *c*. Сам по себе алгоритм *merge* работает в режиме замещения, то есть не создает новых элементов контейнера, а помещает значения в существующие. Чтобы вставлять новые элементы при объединении, мы должны использовать вставляющий итератор, как показано в полной программе. В любом случае результат работы программы следующий:

```
2 3 7 8 9 20 23 25 28 30 33
```

Как и в разделе 1.4, мы использовали операцию доступа по индексу, написав, например, *a*[0], хотя *a* является вектором, а не массивом. Такое же обозначение доступа к элементу используется для двусторонних очередей, но не для списков. Напомним, что списки отличаются от массивов, векторов и двусторонних очередей тем, что мы не можем использовать их вместе с алгоритмом сортировки. К этому вопросу мы вернемся в разделе 1.9.

1.8. Типы, определенные пользователем

До сих пор все контейнеры, которые мы использовали, содержали элементы типа *int*. Кроме стандартных типов, таких как *int*, в контейнерах STL можно хранить типы, определенные пользователем. Так как вызов *merge*(...) в программе *merge.cpp* основан на операции сравнения «меньше чем» <, такой вызов для новых типов возможен, только если мы для этих типов определяем *operator*<. Покажем это на простом примере:

```
// merge2.cpp: Объединяем записи, используя имена
//             в качестве ключей.

#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

struct entry {
   long nr;
   char name[30];
   bool operator<(const entry &b)const
   {  return strcmp(name, b.name) < 0;
   }
};

int main()
{  entry a[3] = {{10, "Betty"},
                 {11, "James"},
                 {80, "Jim"}},
         b[2] =      {{16, "Fred"},
                 {20, "William"}},
         c[5], *p;
   merge(a, a+3, b, b+2, c);
   for (p=c; p != c+5; p++)
      cout << p->nr << " " << p->name << endl;
   cout << endl;
   return 0;
}
```

Для работы программы существенно, что имена в каждом из массивов *a* и *b* располагаются в алфавитном порядке. Программа объединяет *a* и *b* в *c* (в соответствии с алфавитным порядком имен), как показывает результат программы:

```
10 Betty
16 Fred
11 James
80 Jim
20 William
```

Если бы мы хотели, чтобы числа шли в порядке возрастания, их нужно было бы перечислить в таком порядке в заданных массивах *a* и *b* и, кроме того, заменить определение оператора «меньше». Поскольку числа и так уже расположены в порядке возрастания в обоих массивах, нам остается только вместо функции-члена *operator<* использовать следующую:

```
bool operator<(const entry &b) const
{  return nr < b.r;
}
```

После этой модификации вывод программы будет содержать числа в восходящем порядке:

```
10 Betty
11 James
16 Fred
20 William
80 Jim
```

Функция *operator<* не обязана быть членом класса *entry*. Иными словами, мы могли бы написать

```
struct entry {
   long nr;
   char name[30];
};

bool operator<( const entry &a, const entry &b) const
{  return strcmp(a.name, b.name) < 0;
}
```

вместо того определения класса *entry*, которое приводится в программе *merge2.cpp*. Кстати, в разделе 7.3.7 мы увидим, что существует версия *merge*, которая позволяет нам задать функцию сравнения в виде аргумента.

1.9. Категории итераторов

Как видно из раздела 1.4, мы можем использовать алгоритм *sort* для массивов, векторов и двусторонних очередей, но не для списков. Алгоритм *find*, напротив, может быть использован для всех четырех типов контейнеров. (В этом разделе воспользуемся термином *контейнер*, имея в виду *последовательный контейнер*, игнорируя при этом другие типы контейнеров, которые обсудим в главах 2 и 4.) Ясно, что для нахождения заданного значения достаточно одного перебора всех элементов, тогда как эффективная сортировка требует наличия произвольного доступа. В обоих случаях используются *итераторы*, но алгоритм *sort* требует «более мощных» итераторов,

нежели алгоритм *find*. Итераторы можно естественным образом поделить на пять категорий, в соответствии с теми операциями, которые для них определены. Предположим, что *i* и *j* – итераторы одного вида. Тогда три следующие операции

```
i == j          i != j       i = j
```

возможны всегда, независимо от категории этих итераторов. Следующая таблица показывает, какие из операций применимы к каждой из категорий итераторов; мы предполагаем, что *x* является переменной того же типа, что и элементы рассматриваемого контейнера, а *n* – переменная типа *int*:

Категория итератора	***Операции* (дополнительно к *i == j, i != j, i = j***	**Какие контейнеры предоставляют**	**Каким алгоритмом используется**
входной	x = *i, ++i, i++	все четыре	find
выходной	*i = x, ++i, i++	все четыре	copy (приемник)
прямой	как у входного и выходного сразу	все четыре	replace
двунаправленный	как у прямого и --i, i--	все четыре	reverse
произвольного доступа	как у двунаправленного и i + n, i - n, i += n, i -= n, i < j, i > j, i <= j, i >= j	array, vector, deque (но не list)	sort

Как указано во втором столбце таблицы, *прямой* (*forward*) итератор поддерживает все операции как *входных* (*input*), так и *выходных* (*output*) итераторов. *Двунаправленные* (*bidirectional*) итераторы в дополнение ко всем операциям, поддерживаемым прямыми итераторами, поддерживают операции префиксного и постфиксного уменьшения. Добавив к этому набор операций +, –, +=, –=, <, >, <=, >=, мы приходим к категории итераторов *произвольного доступа* (*random access*). Добавление целого числа к итератору возможно только для итераторов произвольного доступа; эта операция требуется, к примеру, для алгоритма сортировки. Так как список не предоставляет итераторов произвольного доступа, мы не можем применить к списку алгоритм *sort*.

Программа *find*1 в разделе 1.5 содержит строчку

```
else cout << " after " << *--i;
```

Мог возникнуть вопрос: почему мы не используем выражение *(*i* – 1) вместо *--*i*, поскольку не было необходимости изменять значение *i*? Теперь мы видим, что выражение *i* – 1 допустимо только для операторов произвольного доступа, тогда как --*i* поддерживается также двунаправленными итераторами. В нынешнем варианте программа *find*1 основана на векторе, а поскольку вектор работает с итераторами произвольного доступа, запись *(*i* – 1) не создаст проблем. Однако проблемы появятся, если слово *vector* везде, где оно встречается в программе, заменить на *list*. В этом случае переменная *i* станет двунаправленным итератором, для которого выражение *i* – 1 не является допустимым. Другими словами, программа *find*1 перестанет компилироваться, если мы одновременно заменим *vector* на *list*, а *--*i* на *(*i* – 1). Точно так же и для итераторов, определенных классом *list*, недоступна операция сравнения «меньше чем». Это иллюстрирует комментарий в следующем фрагменте:

```
// Демонстрация итераторов произвольного доступа:
int a[3] = {5, 8, 2};
vector<int> v(a, a+3);
vector<int>::iterator iv = v.begin(), iv1;

iv1 = iv + 1;
bool b1 = iv < iv1;
// В двух последних строчках + и < допустимы, поскольку
// iv и iv1 - итераторы произвольного доступа.

// Демонстрация двунаправленных итераторов:
list<int> w(a, a+3);
list<int>::iterator iw = w.begin(), iw1;

iw1 = iw + 1;          // Ошибка
bool b2 = iw < iw1;    // Ошибка
// В двух последних строчках + и < недопустимы, поскольку
// iw и iw1 -  двунаправленные итераторы.
// Следующие две строчки, напротив, являются правильными:
iw1 = iw;
bool b3 = iw == iw1;
```

Какие категории итераторов требуются для алгоритмов

Алгоритм *find* (рассмотренный в разделе 1.5) из всех операций над итераторами требует исключительно те, которые определены для входных итераторов, потому что ему достаточно только читать элементы последовательности, исполняя, например, операцию присваивания *x* = **i*. Поэтому в

вышеприведенной таблице *find* служит примером для входных итераторов в столбце *Каким алгоритмом используется*.

Вспомним, как мы использовали алгоритм *copy* (также упомянутый в рассмотренной таблице) в разделе 1.6:

```
copy(v.begin(), v.end(), L.begin());
```

Чтобы увидеть, как это согласуется с нашей таблицей, заметим, что приемник копирования *L* является списком, который определяет итераторы, относящиеся к двунаправленной категории. Для *L* достаточно использовать выходные итераторы, но двунаправленные итераторы поддерживают все операции выходных итераторов, так что проблем не возникает. Так как алгоритм *copy* требует выходной итератор в качестве третьего аргумента, он не будет применять операторы ––, +, –, <, <=, >= и > к итераторам, ссылающимся на элементы списка *L*; также он не будет читать что-либо из *L*, исполняя операции вроде $x = *i$. Однако он воспользуется противоположной операцией $*i = x$, чтобы записывать копируемые значения в *L*.

Потоковые итераторы: использование функции *copy* для ввода и вывода

Мы можем применить алгоритм *copy* для вывода, как показано в следующем фрагменте:

```
const int N = 4;
int a[N] = {7, 6, 9, 2};
copy(a, a+N, ostream_iterator<int>(cout, " "));
```

Мы также можем определить переменную-итератор *i* для использования в качестве третьего аргумента функции *copy*. Это достигается заменой вызова *copy* следующими двумя операторами:

```
ostream_iterator<int> i(cout, " ");
copy(a, a+N, i);
```

В любом случае в поток стандартного вывода *cout* будут выведены числа 7, 6, 9 и 2, как если бы мы написали

```
for (int* p=a; p!= a+N; p++)
   cout << *p << " ";
```

Для ввода мы можем использовать аналогичный прием, написав

```
istream_iterator<int, ptrdiff_t>(file)
```

где *file* является потоком ввода. Это длинное выражение можно, например, употребить вместо *v.begin()*, которое мы использовали бы для копирования элементов из контейнера *v*. Если опустить *file* в приведенной строчке, то получим выражение

```
istream_iterator<int, ptrdiff_t>()
```

которое нужно применить вместо *v.end*(),так как это выражение служит для обозначения конца файла. Например, пусть у нас есть файл *example.txt* со следующим содержанием:

```
10 20 30
40 50
```

Известно, что файл содержит только целые числа (разделенные пробелами), но мы не знаем заранее, сколько целых чисел находится в файле. Следующая программа читает целые числа из этого файла и показывает их на экране; условная компиляция используется для совместимости с BC5, который требует наличия второго аргумента для шаблона *istream_iterator*:

```
// copyio.cpp: Используем алгоритм copy
//               для ввода-вывода.

#include <fstream>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

#if defined(__BORLANDC__) && __BORLANDC__ == 0x530
                                // Для BC5.3:
typedef istream_iterator<int,
                         char,
                         char_traits<char>,
                         ptrdiff_t>
   istream_iter;
#elif defined(__BORLANDC__)     // Для BC5.2:

typedef istream_iterator<int, ptrdiff_t> istream_iter;
#else                           // Для VC5.0:
typedef istream_iterator<int> istream_iter;
#endif

int main()
{  vector<int> a;

   ifstream file("example.txt");
   if (file.fail())
   {  cout << "Cannot open file example.txt.\n";
      return 1;
   }
   copy(istream_iter(file), istream_iter(),
      inserter(a, a.begin()));
   copy(a.begin(), a.end(),
```

```
#if defined(__BORLANDC__) && __BORLANDC__ == 0x530
      ostream_iterator<int,
                       char,
                       char_traits<char> >(cout, " "));
#else
      ostream_iterator<int>(cout, " "));
#endif
   cout << endl;
   return 0;
}
```

Поскольку эта программа не принимает во внимание структуру входного файла, вывод состоит только из одной строчки:

```
10 20 30 40 50
```

Мы вернемся к потоковым итераторам в разделе 6.6.

Операции с итераторами

Вспомним, что мы можем применять арифметические операции к итераторам произвольного доступа, подобно тому, как оперируем с указателями, написав, например:

```
int n, dist;
...      // i и i0 являются итераторами произвольного доступа
i0 = i;
i += n;
dist = i - i0;   // dist == n
```

Вместо этого мы можем использовать функции *advance* и *distance*:

```
int n, dist;
...      // i и i0 являются итераторами, но
         // необязательно произвольного доступа
i0 = i;
advance(i, n);
dist = 0;
distance(i0, i, dist); // dist == n
```

Приведенный фрагмент[1] работает для всех итераторов, если *n* имеет разумное значение, так что модифицированное значение *i* ссылается либо на существующий, либо на следующий за последним элемент рассматриваемого контейнера. Если *i* является прямым итератором, *n* должно быть

[1] В стандарте C++ принята функция *distance* с двумя аргументами вместо трех. Поэтому для компилятора, отвечающего стандарту, следует писать: *dist* = *distance*(*i*0, *i*) вместо *dist* = 0; *distance*(*i*0, *i*, *dist*). – *Прим. переводчика.*

положительным. Тогда приведенный выше вызов функции *advance* имеет то же действие, что и применение оператора ++ к *i* *n* раз. Операции *advance* и *distance* будут выполняться гораздо быстрее для итераторов произвольного доступа, чем для итераторов других типов.

1.10. Алгоритмы *replace* и *reverse*

Алгоритм *replace*, упомянутый в таблице раздела 1.9, позволяет нам найти все элементы с определенным значением в заданном контейнере и заменить их другим значением. Следующая программа служит иллюстрацией сказанного:

```
// replace.cpp: Замена элементов последовательности.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{  char str[] = "abcabcabc";
   int n = strlen(str);
   replace(str, str+n, 'b', 'q');
   cout << str << endl;
   return 0;
}
```

Программа заменяет все элементы массива *str*, равные '*b*', на '*q*', так что вывод этой программы будет следующим:

```
aqcaqcaqc
```

Алгоритм *reverse*, также упомянутый в таблице из раздела 1.9, позволяет легко заменить последовательность на обратную ей. Согласно этой таблице он требует двунаправленных итераторов, которые, как мы знаем, предоставляют все четыре контейнера. Давайте продемонстрируем этот алгоритм снова с использованием массива.

```
// reverse.cpp: Замена строки на обратную ей.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{  char str[] = "abcklmxyz";
   reverse(str, str+strlen(str));
```

```
    cout << str << endl;  // Будет выведено: zyxmlkcba
    return 0;
}
```

1.11. Возвращаясь к алгоритму *sort*

Функция *qsort* из стандартной библиотеки C является достаточно общей, потому что она принимает в качестве четвертого аргумента функцию сравнения, определяемую пользователем. При использовании алгоритма *sort* в разделе 1.4 подобный аргумент отсутствовал, так как *sort* полагался на оператор сравнения «меньше чем» <. Если имеем дело с массивами экземпляров класса, мы можем придать этому оператору любое значение по нашему усмотрению, но это невозможно, если элементами массива являются значения какого-либо встроенного типа, например *int*. Допустим, мы захотим отсортировать массив в нисходящем, а не в восходящем порядке. Конечно, это можно сделать, отсортировав массив сначала в восходящем порядке, а затем применив алгоритм *reverse*, обсуждавшийся в предыдущем разделе. Но мы можем решить задачу другим способом, используя алгоритм *sort* с третьим аргументом, который задает функцию сравнения, как в случае с *qsort*. Например:

```
// dsort1.cpp: Сортировка в нисходящем порядке
//             с использованием функции сравнения.
#include <iostream>
#include <algorithm>
using namespace std;

bool comparefun(int x, int y)
{  return x > y;
}

int main()
{  const int N = 8;
   int a[N] =
      {1234, 5432, 8943, 3346, 9831, 7842, 8863, 9820};
   cout << "Before sorting:\n";
   copy(a, a+N, ostream_iterator<int>(cout, " "));
   cout << endl;
   sort(a, a+N, comparefun);
   cout << "After sorting in descending order:\n";
   copy(a, a+N, ostream_iterator<int>(cout, " "));
   cout << endl;
   return 0;
}
```

Вспомним, что мы обсуждали использование *copy* для вывода, как это делается в данной программе, в конце раздела 1.9. Нам необходимо определить функцию *comparefun* таким образом, чтобы значение *comparefun*(*a*[*i*], *a*[*j*]) равнялось *true* тогда и только тогда, когда после сортировки *a*[*i*] должно предшествовать *a*[*j*]. Функции, которые подобно *comparefun* возвращают значение типа *bool*, называются *предикатами*.

Программа создает следующий вывод:

```
Before sorting:
1234 5432 8943 3346 9831 7842 8863 9820
After sorting in descending order:
9831 9820 8943 8863 7842 5432 3346 1234
```

1.12. Введение в функциональные объекты

Существует другой способ решения задачи сортировки из предыдущего раздела. Хотя для такой простой задачи он и не нужен, обсуждаемые принципы являются важными для других более сложных случаев, поэтому не стоит пропускать этот раздел при чтении. *Функциональным объектом* называется класс, где определен оператор вызова, который записывается как *operator*(). От класса не требуется наличия каких-либо других членов. Давайте начнем с очень простого примера. (Здесь и далее мы пишем *iostream.h* вместо *iostream*, поскольку в последнем случае VC5 также требует наличия строчки *using namespace std*, а BC5, напротив, не позволяет использовать эту строчку, если не включаются типичные заголовки STL, такие как *vector*.)

```
// funobj.cpp: Очень простой функциональный объект.
#include <iostream.h>

class compare {
public:
   int operator()(int x, int y)const
   {  return x > y;
   }
};

int main()
{  compare v;
   cout << v(2, 15) << endl;         // Вывод: 0
   cout << compare()(5, 3) << endl; // Вывод: 1
   cout << endl;
   return 0;
}
```

Так как для класса *compare* определен оператор вызова функции, *operator*(), с двумя параметрами типа *int*, мы можем использовать выражение *v*(2, 15), где

v – переменная этого класса. Это выражение на самом деле является сокращенной формой записи вместо *v.operator*()(2, 15) и таким образом приводит к вызову функции-члена *operator*() класса *compare*, возвращающей значение 0, поскольку 2 не больше 15. Второй вызов

```
compare()(5, 3)
```

выглядит довольно необычно. Первая часть этой записи, *compare*(), представляет собой вызов конструктора по умолчанию класса *compare*. Другими словами, выражение *compare*() представляет объект типа *compare*, и за ним, как и у *v*, может следовать список аргументов, в этом примере (5, 3).

Следующая программа, основывающаяся на использовании функционального объекта, эквивалентна программе *dsort1* из предыдущего раздела:

```
// dsort2.cpp: Сортировка в нисходящем порядке
//             с использованием определенного нами
//             функционального объекта.
#include <iostream>
#include <algorithm>
using namespace std;
class compare {
public:
   bool operator()(int x, int y)const
   {  return x > y;
   }
};

int main()
{  const int N = 8;
   int a[N] =
      {1234, 5432, 8943, 3346, 9831, 7842, 8863, 9820};
   cout << "Before sorting:\n";
   copy(a, a+N, ostream_iterator<int>(cout, " "));
   cout << endl;
   sort(a, a+N, compare());
   cout << "After sorting in descending order:\n";
   copy(a, a+N, ostream_iterator<int>(cout, " "));
   cout << endl;
   return 0;
}
```

В действительности нет необходимости определять класс *compare*, поскольку в STL уже есть подобное определение в более общей форме шаблона. Следовательно, мы можем опустить класс *compare* и при вызове алгоритма *sort* заменить *compare*() на *greater<int>*(). Это дает нам следующую, окончательную версию:

```
// dsort3.cpp: Сортировка в нисходящем порядке
//             с использованием шаблона 'greater'.
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{  const int N = 8;
   int a[N] =
      {1234, 5432, 8943, 3346, 9831, 7842, 8863, 9820};
   cout << "Before sorting:\n";
   copy(a, a+N, ostream_iterator<int>(cout, " "));
   cout << endl;
   sort(a, a+N, greater<int>());
   cout << "After sorting in descending order:\n";
   copy(a, a+N, ostream_iterator<int>(cout, " "));
   cout << endl;
   return 0;
}
```

Обсудим функциональные объекты более подробно в главе 6.

1.13. Использование *find_if*, *remove* и *remove_if*

В этом разделе мы рассмотрим три алгоритма, используя их для векторов, но имея в виду, что они также применимы к двусторонним очередям и спискам.

Алгоритм *find_if*

В дополнение к алгоритму *find*, использовавшемуся в разделе 1.5 для нахождения элементов вектора с заданным значением, имеется также алгоритм *find_if*, который является более общим в том смысле, что в качестве одного из аргументов принимает предикат (см. раздел 1.11). Этот алгоритм ищет в векторе первый элемент, который удовлетворяет условию, указанному в этом предикате. Например, следующая программа ищет в векторе первый элемент $*i$, удовлетворяющий

$$3 \leqslant *i \leqslant 8$$

Эта программа проверяет, найден ли такой элемент, и в этом случае печатает его значение:

```
// find_if.cpp: Демонстрация алгоритма find_if.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
bool condition(int x)
{  return 3 <= x && x <= 8;
}

int main()
{  vector<int> v;
   v.push_back(10); v.push_back(7);
   v.push_back(4); v.push_back(1);
   vector<int>::iterator i;
   i = find_if(v.begin(), v.end(), condition);
   if (i != v.end())
      cout << "Found element: " << *i << endl;
   return 0;
}
```

Как легко предугадать, вывод программы будет следующий:

```
Found element: 7
```

поскольку в последовательности {10, 7, 4, 1} значение 7 первое находится в диапазоне между 3 и 8. Программа также будет правильно работать, если заменить функцию функциональным объектом (см. раздел 1.12). Для этого вместо функции *condition* напишем следующее определение класса:

```
class CondObject {
public:
   bool operator()(int x)
   {  return 3 <= x && x <= 8;
   }
};
```

В то же время мы заменим вызов *find_if* в тексте программы на следующий:

```
i = find_if(v.begin(), v.end(), CondObject());
```

Алгоритмы *remove* и *remove_if*

Предположим, нам требуется удалить все элементы вектора, равные определенному значению. Это можно сделать, несколько раз подряд вызывая *find* и *erase*, но существует и более эффективный способ добиться того же самого. Необходимо выполнить два действия:

1. Переупорядочить вектор с помощью вызова *remove*, разместив все элементы, которые мы хотим оставить, в начале вектора. Эта операция *стабильна*: порядок, в котором будут находиться сохраненные элементы, не изменится.

2. Стереть те элементы, которые нам не нужны; теперь они расположены в конце вектора.

Алгоритм *remove* возвращает новый логический конец данных, не изменяя размер контейнера. Мы можем стереть «мусор», начиная с возвращенного конца с помощью функции-члена *erase*, рассмотренной в конце раздела 1.3. Следующая программа показывает, как это делается, удаляя все элементы, равные 1, из последовательности:

```
// remove.cpp: Алгоритм remove.

#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

void out(const char *s, const vector<int> &v)
{  cout << s;
   copy(v.begin(), v.end(),
      ostream_iterator<int>(cout, " "));
   cout << endl;
}

int main()
{  vector<int> v;
   vector<int>::iterator new_end;
   v.push_back(1); v.push_back(4); v.push_back(1);
   v.push_back(3); v.push_back(1); v.push_back(2);
   out("Initial sequence v:\n", v);
   new_end = remove(v.begin(), v.end(), 1);
   out("After new_end = remove(v.begin(), "
      "v.end(), 1):\n", v);
   v.erase(new_end, v.end());
   out("After v.erase(new_end, v.end()):\n", v);
   return 0;
}
```

Начиная с последовательности {1, 4, 1, 3, 1, 2}, эта программа логически удаляет элементы, равные 1; остающиеся элементы помещаются в первых

$$new\_end - v.begin = 3$$

элементах, где *new_end* – значение, возвращаемое вызовом функции *remove*. Остальные три элемента, начиная с позиции *new_end*, затем стираются с помощью функции-члена *erase*. Вывод этой программы показан ниже:

```
Initial sequence v:
1 4 1 3 1 2
After new_end = remove(v.begin(), v.end(), 1):
4 3 2 3 1 2
After v.erase(new_end, v.end()):
4 3 2
```

Стабильность алгоритма *remove* иллюстрируется порядком 4, 3, 2 оставшихся элементов – этот порядок совпадает с тем, в котором эти элементы встречались в исходной последовательности. Хотя алгоритм *remove* применим также и к спискам, для них определена функция-член *remove*, использование которой предпочтительнее, как мы увидим в разделе 3.5.

Существует также алгоритм *remove_if*, являющийся обобщенной версией *remove*, точно так же как *find_if* – обобщенная версия *find*. Следующая программа показывает, как мы можем использовать *remove_if* (опять совместно с функцией-членом *erase*), чтобы стереть все элементы, меньшие или равные 2:

```
// rm_if.cpp: Алгоритм remove_if.

#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

void out(const char *s, const vector<int> &v)
{  cout << s;
   copy(v.begin(), v.end(),
      ostream_iterator<int>(cout, " "));
   cout << endl;
}

bool cond(int x)
{  return x <= 2;
}

int main()
{  vector<int> v;
   vector<int>::iterator new_end;
   v.push_back(1); v.push_back(4); v.push_back(1);
   v.push_back(3); v.push_back(1); v.push_back(2);
   out("Initial sequence v:\n", v);
   new_end = remove_if(v.begin(), v.end(), cond);
   v.erase(new_end, v.end());
   out("After erasing all elements <= 2:\n", v);
   return 0;
}
```

Эта программа выдает:

```
Initial sequence v:
1 4 1 3 1 2
After erasing all elements <= 2:
4 3
```

Как и *remove*, алгоритм *remove_if* является стабильным, поэтому совпадение порядка следования элементов 4 и 3 в результирующей последовательности с их порядком в исходной последовательности не является случайным.

1.14. Класс *auto_ptr*

После того как размещена динамическая память, нужно внимательно следить за тем, чтобы она была правильно освобождена. Обычно программисты на C используют для этого *malloc* и *free*, а программисты на C++ используют также *new* и *delete*. Первое, о чем необходимо помнить, что после

```
int *p = new int;
int *q = new int[n];
```

в дальнейшем требуется выполнить следующие операторы:

```
delete p;
delete[] q;
```

Пара квадратных скобок [] должна использоваться для q, но не для p; несоблюдение этого правила – довольно частая причина ошибок. Другая сложность заключается в необходимости отслеживать копии указателей. Например, если мы напишем

```
int *a = new int[n], *b;
b = a;
```

a и *b* будут указывать на один и тот же блок памяти. Освобождение этого блока должно быть проведено только один раз, поэтому позже мы должны выполнить либо

```
delete[] a;
```

либо

```
delete[] b;
```

но не оба эти выражения.

В STL определен специальный класс *auto_ptr*, который делает более безопасным использование оператора *new* в сложных случаях. Функция-член *get* возвращает указатель на сами данные. Если мы присвоим

значение *auto_ptr a* переменной *b*, указателю *a.get*() автоматически будет присвоено значение *NULL*, например:

```
// auto_ptr.cpp: Данным соответствует
// только один указатель.
#include <iostream>
#include <memory>
using namespace std;

int main()
{  auto_ptr<int> a(new int), b;
   *a.get() = 123;
   cout << "*a.get() = " << *a.get() << endl;
   b = a;
   cout << "The assignment b = a has been executed.\n";
   if (a.get() == NULL)
      cout << "As a result, a.get() is NULL.\n";
   cout << "*b.get() = " << *b.get() << endl;
   return 0;
}
```

Вывод этой программы следующий:

```
*a.get() = 123
The assignment b = a has been executed.
As a result, a.get() is NULL.
*b.get() = 123
```

Поскольку выражение *a.get*() является обычным указателем, значение, на которое оно указывает, обозначается как **a.get*(). Преимущество использования класса *auto_ptr* заключается в том, что память освобождается автоматически, когда уничтожается объект, принадлежащий этому классу, и это происходит только один раз: в нашем примере переменные *a* и *b* не содержат одновременно указатели на одну и ту же область памяти. Для прояснения механизма действия класса *auto_ptr* посмотрим на его деструктор (в этом коде *the_p* – скрытый (private) член класса, указывающий на размещенные данные):

```
~auto_ptr(){delete the_p;}
```

В нашей программе деструктор вызывается как для *a*, так и для *b*, поэтому выполняются оба действия

```
delete a.the_p;
delete b.the_p;
```

что может показаться ошибкой. Однако один из двух используемых здесь указателей равен *NULL*, а применение оператора *delete* к указателю, равно-

му *NULL*, является разрешенной операцией, которая не вызывает никаких действий.

Следует упомянуть об унарном операторе *, который возвращает **the_p*. Поскольку функция-член *get* возвращает указатель *the_p*, мы упростим использованную нами запись. Например, вместо

```
*a.get() = 123;
```

напишем

```
*a = 123;
```

но должны иметь в виду, что *a* не является настоящим указателем.

Предупреждение

В отличие от прочих составляющих библиотеки STL класс *auto_ptr* иногда критикуют за странное поведение, поэтому используйте его только тогда, когда точно знаете, что вы делаете. Например, после выполнения

```
auto_ptr<int> a(new int), b;
*a.get() = 123;
b = a;
```

может показаться странным, что, хотя значение **b* определено и равно 123, мы не имеем права использовать выражение **a*, поскольку последнее присваивание приводит к обнулению указателя на целое, содержащегося в *a*.

Причина этого заключена в необходимости освобождать память, выделенную с помощью оператора *new*, ровно один раз.

2

Другие алгоритмы и контейнеры

2.1. Алгоритм *accumulate*

Нахождение суммы элементов последовательности или подпоследовательности лучше всего достигается с помощью алгоритма *accumulate*. Этот алгоритм вместе с некоторыми другими, которые имеют отношение к вычислениям, определен в заголовке *numeric*, а не *algorithm*, как большинство остальных алгоритмов. (Если вы работаете с HP STL, то должны использовать *algo.h* вместо *algorithm* или *numeric*.) Следующая программа показывает, как использовать алгоритм *accumulate* для массива:

```
// accum1.cpp: Вычисление сумм.
#include <iostream>
#include <numeric>
using namespace std;

int main()
{  const int N = 8;
   int a[N] = {4, 12, 3, 6, 10, 7, 8, 5}, sum = 0;
   sum = accumulate(a, a+N, sum);
   cout  << "Sum of all elements: " << sum << endl;
   cout  << "1000 + a[2] + a[3] + a[4] = "
         << accumulate(a+2, a+5, 1000) << endl;
   return 0;
}
```

Первый и второй аргументы алгоритма *accumulate* указывают последовательность, сумму элементов которой мы хотим вычислить. Третий аргумент задает начальное значение для процесса суммирования и, следовательно, обычно равен нулю. При первом вызове *accumulate* мы могли бы использовать константу 0 вместо переменной *sum* в качестве третьего аргумента. Вот результат работы этой программы:

```
Sum of all elements: 55
1000 + a[2] + a[3] + a[4] = 1019
```

Эти значения вычисляются как

$0 + 4 + 12 + 3 + 6 + 10 + 7 + 8 + 5 = 55$
$1000 + 3 + 6 + 10 = 1019$

Шаблон *multiplies<int>*() аналогичен рассмотренному в конце раздела 1.12 шаблону *greater<int>*(). Мы используем его для вычисления произведения вместо суммы:

```
// accum2.cpp: Вычисление произведения.
#include <iostream>
#include <numeric>
#include <algorithm>
#include <functional>

using namespace std;
int main()
{  const int N = 4;
   int a[N] = {2, 10, 5, 3}, prod = 1;
   prod = accumulate(a, a+N, prod, multiplies<int>());
   // ('muliplies' бывший 'times')
   cout << "Product of all elements: " << prod << endl;
   return 0;
}
```

Заметим, что ранее шаблон назывался *times*, а не *multiplies*. Это по-прежнему так для BC 5.2. Вывод программы составляет 300 ($= 1 \times 2 \times 10 \times 5 \times 3$). В этом примере существенно, что третий аргумент алгоритма равен 1 (это «идентичное» или «нейтральное» значение для умножения). Четвертый аргумент указывает, что будет вычисляться целочисленное произведение.

Снова обратимся к алгоритму *accumulate*, на этот раз используя наш собственный функциональный объект. Для заданного массива *a*, содержащего, скажем, четыре элемента, вычисляется следующее значение:

```
1 * a[0] + 2 * a[1] + 4 * a[2] + 8 * a[3]
```

Кроме функции *operator*() наш функциональный объект содержит член типа *int*, который хранит последовательные степени 1, 2, 4 и 8, а также конструктор для инициализации этого члена класса:

```
// accum3.cpp: Вычисление следующей суммы:
//             1 * a[0] + 2 * a[1] + 4 * a[2] + 8 * a[3].
#include <iostream>
#include <numeric>

using namespace std;

class fun {
public:
   fun(){i = 1;}
   int operator()(int x, int y)
   {  int u = x + i * y;
      i *= 2;
      return u;
   }
private:
   int i;
};

int main()
{  const int N = 4;
   int a[N] = {7, 6, 9, 2}, prod = 0;
   prod = accumulate(a, a+N, prod, fun());
   cout << prod << endl;
   return 0;
}
```

Эта программа выведет значение 71 (= $1 \times 7 + 2 \times 6 + 4 \times 9 + 8 \times 2$).

2.2. Алгоритм *for_each*

Мы можем использовать алгоритм *for_each* для вызова функции с каждым из элементов последовательности в качестве аргумента. Вот программа, которая демонстрирует это:

```
// for_each.cpp: Алгоритм for_each.
#include <iostream>
#include <algorithm>

using namespace std;

void display(int x)
{  static int i=0;
   cout << "a[" << i++ << "] = " << x << endl;
}

int main()
{  const int N = 4;
```

```
    int a[N] =
       {1234, 5432, 8943, 3346};
    for_each(a, a+N, display);
    return 0;
}
```

Эта программа работает точно так же, как если бы мы заменили вызов *for_each* следующим оператором *for*:

```
for (int *p=a; p != a+N; p++)
   display(*p);
```

В любом случае программа выведет

```
a[0] = 1234
a[1] = 5432
a[2] = 8943
a[3] = 3346
```

Функция *display* в этом примере обладает существенным недостатком: переменная *i* равна нулю только тогда, когда эта функция вызывается в первый раз. Например, еще один такой же вызов *for_each* в функции *main* приведет к ошибке, поскольку при повторном вызове функция *display* не начнет с *i* = 0. Мы можем решить эту проблему с помощью функционального объекта. Заменим функцию *display* следующим определением класса:

```
class display {
public:
   display(): i(0){}
   void operator()(int x)
   {  cout << "a[" << i++ << "] = " << x << endl;
   }
private:
   int i;
};
```

Также добавим пару скобок к третьему аргументу *for_each*:

```
for_each(a, a+N, display());
```

С этими изменениями программа выдаст прежние результаты. В отличие от первоначальной программы этот вывод будет повторен дважды, если мы напишем два вызова *for_each*.

2.3. Подсчет

Алгоритм *count* подсчитывает, какое количество элементов последовательности равно заданному значению. Давайте используем этот алгоритм для того, чтобы подсчитать, сколько раз в строке встречается буква *e*.

```
// count_e.cpp: Подсчет количества букв 'e'.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main() // Для BC 5.2 требуются изменения (см. ниже)
{  char *p =
      "This demonstrates the Standard Template Library";
   int n = count(p, p + strlen(p), 'e');
   cout << n << " occurrences of 'e' found.\n";
   return 0;
}
```

В изначальной версии STL алгоритм *count* не возвращал значение, а использовал четвертый параметр, который увеличивался на найденную величину. Это по-прежнему так для BC 5.2.

Следующая программа подсчитывает, сколько символов из множества {'*a*', '*e*', '*i*', '*o*', '*u*'} (так называемые гласные) встречаются в заданной строке.

```
// countvw1.cpp: Сосчитать, сколько раз гласные
//               a, e, i, o, u встречаются в заданной строке
//               (первая версия).
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{  char *p =
      "This demonstrates the Standard Template Library",
      *q = p + strlen(p);
   int n = count(p, q, 'a') +
      count(p, q, 'e') +
      count(p, q, 'i') +
      count(p, q, 'o') +
      count(p, q, 'u');
   cout << n << " vowels (a, e, i, o, u) found.\n";
      // n = 13
   return 0;
}
```

К сожалению, этот подход не слишком эффективен, так как заданная строка сканируется пять раз, а мы бы предпочли один проход. Это достигается с помощью алгоритма *count_if,* которому можно передать функцию, определяющую, удовлетворяется ли требуемое условие. Число параметров функции *count_if* недавно уменьшилось на один, точно так же, как и у *count*.

```
// countvw2.cpp: Сосчитать, сколько раз гласные
//               a, e, i, o, u встречаются в заданной строке
//               (улучшенная версия).
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

bool found(char ch)
{  return ch == 'a' || ch == 'e' || ch == 'i' ||
          ch == 'o' || ch == 'u';
}

int main()
{  char *p =
      "This demonstrates the Standard Template Library";
   int n = count_if(p, p + strlen(p), found);
   cout << n << " vowels (a, e, i, o, u) found.\n";
      // n = 13
   return 0;
}
```

2.4. Функциональные объекты, определенные в STL

Напомним, что такие функции, как *found* (в предыдущем разделе), называются *предикатами*. Они возвращают *true* или *false* в зависимости от соблюдения некоторого условия. Выражение *greater<int>*, которое встретилось нам в разделе 1.11, также является предикатом, но определенным в STL в виде шаблона. Напомним также, что для сортировки последовательности в нисходящем порядке мы использовали это выражение в вызове

```
sort(a, a+N, greater<int>());
```

В разделе 2.1 мы применяли похожее выражение *multiplies<int>*() в следующем вызове, чтобы указать необходимость выполнения умножения:

```
int prod = accumulate(a, a+N, 1, multiplies<int>());
```

Ниже дан полный список таких шаблонов (определенных в заголовке *functional*), соответствующих стандартным бинарным операциям:

```
plus<T>             minus<T>
multiplies<T>       divides<T>              modulus<T>
equal_to<T>         not_equal_to<T>
greater<T>          less<T>
greater_equal<T>    less_equal<T>
logical_and<T>      logical_or<T>
```

Как известно, выражения типа *plus<T>*() являются объектами, другими словами, вышеперечисленные шаблоны, сопровожденные парой круглых скобок,– стандартные функциональные объекты, определенные в библиотеке STL. Кроме этого, существуют шаблоны, соответствующие унарным операторам – (как в выражении –*x*) и ! (произносится *не*):

```
negate<T>           logical_not<T>
```

Теперь вернемся к программе *countvw2.cpp* в конце предыдущего раздела. Она содержит вызов

```
int n = count_if(p, p + strlen(p), found);
```

где *found* – определенная нами функция, указывающая, какие символы необходимо подсчитывать. Предположим, что теперь мы хотим подсчитать все символы *ch* >= 'k'. Очевидно, мы могли бы заменить *found* на следующую функцию:

```
bool found(char ch)
{  return ch >= 'k';
}
```

Вместо этого возможно использовать упомянутый шаблон *greater_equal<T>*, но нам нужен способ связать *greater_equal<char>* со значением '*k*'. Это достигается с помощью записи

```
bind2nd(greater_equal<char>(), 'k')
```

Указанное выражение может заменить имя функции *found* в рассматриваемом вызове *count_if*, что дает

```
n = count_if(p, p + strlen(p),
   bind2nd(greater_equal<char>(), 'k'));
```

Шаблон *bind2nd* называется *привязкой* (являющейся разновидностью *адаптера функции*). Поскольку два выражения

```
ch >= 'k'   и  !(ch < 'k')
```

эквивалентны, нас может заинтересовать, допустимо ли использовать некий аналог второго выражения в качестве третьего аргумента функции *count_if*. Это действительно можно сделать, если применить другой тип адаптера функции, называемый *отрицателем*:

```
n = count_if(p, p + strlen(p),
   not1(bind2nd(less<char>(), 'k')));
```

Мы обсудим адаптеры функций (и прочие адаптеры) более подробно в главе 6.

2.5. Введение в ассоциативные контейнеры

Кроме массивов и списков, использующихся для реализации последовательных контейнеров (массивов, векторов, двусторонних очередей и списков), которые обсуждались до сих пор, сбалансированные деревья представляют собой другую классическую структуру данных, предназначенную для их эффективного хранения и извлечения. Сбалансированные деревья составляют основу для другой группы контейнеров, определенной в STL, так называемых (*сортированных*) *ассоциативных контейнеров*. Как и ранее, мы в основном сосредоточимся на том, *как использовать* эти контейнеры, а не на том, как они реализованы. Всего существует четыре типа этих контейнеров: множества (sets), множества с дубликатами (multisets), словари (maps), словари с дубликатами (multimaps). Перед тем как обсудить их использование, посмотрим, чем они различаются.

Множества

Каждый элемент *множества* является собственным ключом, и эти ключи уникальны. Поэтому два различных элемента множества не могут совпадать. Например, множество может состоять из следующих элементов:

```
123
124
800
950
```

Множества с дубликатами

Множество с дубликатами отличается от просто множества только тем, что способно содержать несколько совпадающих элементов. Например, допустимо существование множества с дубликатами, в котором присутствуют следующие четыре элемента:

```
123
123
800
950
```

Словари

Каждый элемент словаря имеет несколько членов, один из которых является ключом. В словаре не может быть двух одинаковых ключей. Приведем пример словаря из четырех элементов, у каждого из которых присутствует целочисленный ключ и буквенные сопутствующие данные:

```
123   John
124   Mary
```

```
800   Alexander
950   Jim
```

Словари с дубликатами

Словарь с дубликатами отличается от просто словаря тем, что в нем разрешены повторяющиеся ключи. Вот, к примеру, словарь с дубликатами, состоящий из четырех элементов (с целочисленными ключами):

```
123   John
123   Mary
800   Alexander
950   Jim
```

В отличие от последовательных контейнеров ассоциативные контейнеры хранят свои элементы отсортированными, вне зависимости от того, каким образом они были добавлены.

Заголовки и переносимость

В изначальной версии HP STL было четыре заголовка, связанных с рассматриваемой темой: *set.h*, *multiset.h*, *map.h* и *multimap.h*. В проекте стандарта C++ остались только два: *set* и *map*. Они используются также для ассоциативных контейнеров с дубликатами.

2.6. Множества и множества с дубликатами

В этом и следующем разделе мы рассмотрим по одной простой программе для каждого из четырех ассоциативных контейнеров: эти разделы покрывают все возможные операции с этими контейнерами не полностью, но они поясняют наиболее важные их характеристики.

Множества

Начнем с двух множеств целых чисел. Хотя элементы добавляются разными способами, получающиеся множества идентичны.

```
// set.cpp: Два идентичных множества,
//          созданных разными способами.
#include <iostream>
#include <set>
using namespace std;

int main()
{  set<int, less<int> > S, T;
   S.insert(10); S.insert(20); S.insert(30);
   S.insert(10);
   T.insert(20); T.insert(30); T.insert(10);
   if (S == T) cout << "Equal sets, containing:\n";
```

```
    for (set<int, less<int> >::iterator i = T.begin();
       i != T.end(); i++)
          cout << *i << " ";
    cout << endl;
    return 0;
}
```

Программа выведет

```
Equal sets, containing:
10 20 30
```

и это показывает, что порядок 20, 30, 10, в котором были добавлены элементы *T,* несущественен; равным образом множество *S* не изменяет добавление элемента 10 во второй раз. Напомним, что ключи уникальны во множествах, но могут повторяться во множествах с дубликатами.

Обратите внимание на запись, с помощью которой определены *S* и *T*:

```
set<int, less<int> > S, T;
```

Предикат

```
less<int>
```

требуется для определения значения выражения $k_1 < k_2$, где k_1 и k_2 являются ключами. Это выглядит странным в текущем примере, когда ключи – целые числа, но стоит напомнить, что множества могут содержать ключи, тип которых определен пользователем. Пробел между двумя закрывающими угловыми скобками в

```
less<int> >
```

необходим, чтобы компилятор не обнаружил в этом фрагменте оператор >>.

Хотя множества и не являются последовательностями, мы можем применять к ним итераторы и функции *begin* и *end,* как видно из этой программы. Данные итераторы являются двунаправленными (см. раздел 1.9): для итератора *i* типа *set<int, less<int> >::iterator* выражения *++i*, *i++*, *––i* и *i––* являются допустимыми, а *i* + *N* и *i* – *N* – нет.

Множества с дубликатами

Следующая программа демонстрирует, что во множествах с дубликатами могут встречаться одинаковые ключи. Для разнообразия в ней вывод осуществляется с помощью функции *copy*, как показано ранее в разделе 1.9:

```
// multiset.cpp: Два множества с дубликатами.
#include <iostream>
#include <set>
using namespace std;
```

```
int main()
{  multiset<int, less<int> > S, T;
   S.insert(10); S.insert(20); S.insert(30);
   S.insert(10);
   T.insert(20); T.insert(30); T.insert(10);
   if (S == T)    cout << "Equal multisets:\n"; else
                  cout << "Unequal multisets:\n";

   cout << "S: ";
   copy(S.begin(), S.end(),
      ostream_iterator<int>(cout, " "));
   cout << endl;
   cout << "T: ";
   copy(T.begin(), T.end(),
      ostream_iterator<int>(cout, " "));
   cout << endl;
   return 0;
}
```

Вывод программы показывает, что ключ 10 дважды встречается во множестве с дубликатами *S*. Поскольку он встречается только один раз в *T*, эти два множества с дубликатами неравны:

```
Unequal multisets:
S: 10 10 20 30
T: 10 20 30
```

2.7. Словари и словари с дубликатами

Словари

Происхождение термина «ассоциативный контейнер» становится ясным, как только мы начинаем рассматривать словари. Например, телефонный справочник связывает (ассоциирует) имена с номерами. Имея заданное имя или *ключ*, мы хотим узнать соответствующий номер. Другими словами, телефонная книга является отображением имен на числа. Если имя *Johnson, J.* соответствует номеру 12345, STL позволяет нам определить словарь *D*, так что мы можем записать следующий оператор для выражения отображения, показанного на второй строчке:

```
D["Johnson, J."] = 12345;
```

"*Johnson, J.*" → 12345

Отметим сходство с обычными массивами, например:

```
a[5] = 'Q';
```

5 → 'Q'

В последнем случае индексами являются значения 0, 1, 2, ..., но на словари это ограничение не распространяется. Нижеприведенная программа показывает, что словарями удобно пользоваться:

```
// map1.cpp: Первая программа со словарями.
#include <iostream>
#include <string>
#include <map>
using namespace std;

class compare {
public:
   bool operator()(const char *s, const char *t)const
   {  return strcmp(s, t) < 0;
   }
};

int main()
{  map<char*, long, compare> D;
   D["Johnson, J."]  = 12345;
   D["Smith, P."]    = 54321;
   D["Shaw, A."]     = 99999;
   D["Atherton, K."] = 11111;
   char GivenName[30];
   cout << "Enter a name: ";
   cin.get(GivenName, 30);
   if (D.find(GivenName) != D.end())
      cout << "The number is " << D[GivenName];
   else
      cout << "Not found.";
   cout << endl;
   return 0;
}
```

В отличие от предыдущего примера программа *map.cpp* содержит определенный нами функциональный объект (функциональные объекты рассмотрены в разделе 1.12). Определение

```
map<char*, long, compare> D;
```

справочника *D* содержит следующие параметры шаблона:

- тип ключа *char**;
- тип сопутствующих данных *long*;
- класс функционального объекта *compare*.

Функция-член *operator*() класса *compare* определяет отношение *меньше* для ключей.

Словари с дубликатами

Следующая программа показывает, что словари с дубликатами могут содержать повторяющиеся ключи:

```
// multimap.cpp: Множество с дубликатами,
//               содержащее одинаковые ключи.
#include <iostream>
#include <string>
#include <map>

using namespace std;

class compare {
public:
   bool operator()(const char *s, const char *t)const
   {  return strcmp(s, t) < 0;
   }
};

typedef multimap<char*, long, compare> mmtype;
int main()
{  mmtype D;
   D.insert(mmtype::value_type("Johnson, J.", 12345));
   D.insert(mmtype::value_type("Smith, P.", 54321));
   D.insert(mmtype::value_type("Johnson, J.", 10000));
   cout << "There are " << D.size() << " elements.\n";
   return 0;
}
```

Программа выведет:

```
There are 3 elements.
```

Оператор доступа по индексу [] не определен для множеств с дубликатами, поэтому мы не можем добавить элемент, написав, к примеру:

```
D["Johnson, J."] = 12345;
```

Вместо этого напишем:

```
D.insert(mmtype::value_type("Johnson, J.", 12345));
```

где *mmtype* на самом деле означает

```
multimap<char*, long, compare>
```

Так как идентификатор *value_type* определен внутри шаблонного класса *multimap*, перед *value_type* здесь требуется написать префикс *mmtype*::. Определение идентификатора *value_type* основано на шаблоне *pair*, который мы сейчас и обсудим.

2.8. Пары и сравнения

Чтобы использовать словари и словари с дубликатами более интересным способом, нам нужно познакомиться с шаблонным классом *pair* (пара), который полезен также и для других целей. Этот класс использует следующая программа:

```
// pairs.cpp: Операции с парами.
#include <iostream>
#include <utility>

using namespace std;
int main()
{  pair<int, double> P(123, 4.5), Q = P;
   Q = make_pair(122, 4.5);
   cout << "P: " << P.first << " " << P.second << endl;
   cout << "Q: " << Q.first << " " << Q.second << endl;
   if (P > Q) cout << "P > Q\n";
   ++Q.first;
   cout << "After ++Q.first: ";
   if (P == Q) cout << "P == Q\n";
   return 0;
}
```

Большинство программ, использующих STL, будут содержать строчку *#include* для заголовка, который уже включает внутри себя *utility*. Например, если в нашей программе есть строчка *#include <vector>*, не нужно добавлять *#include <utility>*.

Как показывает эта программа, шаблон *pair* имеет два параметра, представляющих собой типы членов структуры *pair: first* и *second*. Конструктор для *pair*, используемый для инициализации *P*, получает два начальных значения для этих членов. У структуры *pair* нет конструктора по умолчанию; другими словами, нельзя написать

```
pair<int, double> P; // ошибка
```

Вместо того чтобы явным образом передавать два значения конструктору, как это сделано для *P*, мы можем использовать уже существующую пару, как показано выше при определении *Q*.

Определив *Q*, можно было бы присвоить этой переменной другое значение, написав

```
Q = pair<int, double>(122, 4.5);
```

Вместо этого мы воспользовались более короткой записью

```
Q = make_pair(122, 4.5);
```

эквивалентной по своему действию. Для любых двух пар *P* и *Q* значения выражений *P* == *Q* и *P* < *Q* и т. п. согласуются с хорошо известным лексикографическим порядком, как показывают следующие примеры:

```
(122, 5.5) < (123, 4.5)
(123, 4.5) < (123, 5.5)
(123, 4.5) == (123, 4.5)
```

В программе *pairs.cpp* сначала мы имеем *P* > *Q*, но после увеличения *Q.first* на единицу *P* и *Q* сравняются, как показывает вывод:

```
P: 123 4.5
Q: 122 4.5
P > Q
After ++Q.first: P == Q
```

Сравнения

Когда мы пишем операторы сравнения для наших собственных типов, нам необходимо определить только == и <. Четыре остающихся оператора !=, >, <= и >= автоматически определяются в STL с помощью следующих четырех шаблонов:

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

template <class T1, class T2>
inline bool operator>(const T1 &x, const T2 &y)
{  return  y < x;
}

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

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

Как видно из примера, эти четыре достаточно общих шаблона определяют !=, >, <= и >= через == и <. Нам не нужно писать эти шаблоны самостоятельно, поскольку они находятся в заголовке *functional*, который включается по умолчанию всякий раз, когда мы используем STL.

2.9. Снова словари

Поскольку словарь содержит пары (*k*, *d*), где *k* является ключом, а *d* – сопутствующими данными, можно предположить, что шаблон *pair* будет полезен при работе со словарями. Как и для последовательного контейнера, для ассоциативного контейнера мы можем использовать итератор *i*; в этом случае выражение **i* будет обозначать пару, в которой (**i*).*first* является ключом, а (**i*).*second* – сопутствующими данными. Например, с помощью итератора *i* для словаря из раздела 2.7 напечатаем все содержимое словаря (ключи в восходящем порядке), применив следующий цикла *for*:

```
for (i = D.begin(); i != D.end(); i++)
   cout  << setw(9)
         << (*i).second <<" "
         << (*i).first << endl;
```

Заметим, что здесь мы выводим (**i*).*second* перед (**i*).*first*, так что не нужно планировать, сколько позиций зарезервировать для имен в выводе вроде

```
54321 Papadimitrou, C.
12345 Smith, J.
```

С таким форматом также удобнее работать при вводе, поскольку в этом случае мы можем прочесть число, пробел и текст до конца строки. Но следует помнить, что этот текст является ключом, хотя и расположен в конце строки.

Приведенный выше цикл *for* действительно встречается в программе, которую мы сейчас собираемся рассмотреть. Она опять имеет отношение к телефонному справочнику, но на этот раз можно произвести с ним несколько операций. Чтобы не усложнять программу, в ее интерфейсе мы будем использовать не самые дружественные к пользователю команд, которые показаны в следующей таблице в виде примеров.

Пример команды	Значение
?Johnson, J.	Показать телефонный номер абонента *Johnson, J.*
/Johnson, J.	Удалить запись об абоненте *Johnson, J.* из книги
!66331 Peterson, K.	Добавить абонента *Peterson, K.* с номером 66331
*****	Показать всю телефонную книгу
=	Записать телефонную книгу в файл *phone.txt*
#	Выход

Когда мы запускаем программу *map2.cpp*, она делает попытку прочитать данные из файла *phone.txt*; такой файл, если он существует, должен содержать строчки текста, включающие номер телефона, один пробел и имя, в перечисленном порядке. В именах могут быть пробелы (см. таблицу выше). Пользователь должен вводить имена в той же форме, как они заданы, вместе с пробелами.

Программа *map2.cpp* основана на обработке строк в стиле языка C. Ключами выступают не сами строки, а указатели на последовательности символов, хранящихся в памяти. Такой подход может удовлетворить опытных программистов на C, но он является не слишком элегантным и требует с нашей стороны внимания, чтобы не допустить ошибочного выделения либо освобождения памяти. В разделе 2.12 мы рассмотрим более простую, хотя, возможно, не столь переносимую версию этой программы, основанную на библиотечном классе *string*.

```
// map2.cpp: Второе приложение, использующее класс
//           map (словарь): телефонный справочник.
#include <iostream>
#include <fstream>
#include <iomanip>
#include <stdlib.h>
#include <string>
#include <map>
using namespace std;
const int maxlen = 200;

class compare {
public:
   bool operator()(const char *s, const char *t)const
   {  return strcmp(s, t) < 0;
   }
};

typedef map<char*, long, compare> directype;

void ReadInput(directype &D)
{  ifstream ifstr("phone.txt");
   long nr;
   char buf[maxlen], *p;
   if (ifstr)
   {  cout << "Entries read from file phone.txt:\n";
      for (;;)
      {  ifstr >> nr;
         ifstr.get();      // пропустить пробел
         ifstr.getline(buf, maxlen);
         if (!ifstr) break;
```

```
            cout << setw(9) << nr << " " << buf << endl;
            p = new char[strlen(buf) + 1];
            strcpy(p, buf);
            D[p] = nr;
         }
      }
      ifstr.close();
}

void ShowCommands()
{  cout <<
      "Commands: ?name       : find phone number,\n"
      "          /name       : delete\n"
      "          !number name: insert (or update)\n"
      "          *           : list whole phonebook\n"
      "          =           : save in file\n"
      "          #           : exit" << endl;
}

void ProcessCommands(directype &D)
{  ofstream ofstr;
   long nr;
   char ch, buf[maxlen], *p;
   directype::iterator i;
   for (;;)
   {  cin >> ch;        // Пропустить любой непечатаемый символ
                        // и прочесть ch.
      switch (ch){
      case '?': case '/':    // найти или удалить:
         cin.getline(buf, maxlen);
         i = D.find(buf);
         if (i == D.end()) cout << "Not found.\n";
         else                    // Ключ найден.
         if (ch == '?')          // Команда 'Найти'
         cout << "Number: " << (*i).second << endl;
         else                    // Команда 'Удалить'
         {  delete[] (*i).first; D.erase(i);
         }
         break;
      case '!':                  // добавить (или обновить)
         cin >> nr;
         if (cin.fail())
         {  cout << "Usage: !number name\n";
            cin.clear(); cin.getline(buf, maxlen);
            break;
         }
         cin.get();              // пропустить пробел;
         cin.getline(buf, maxlen);
```

```
            i = D.find(buf);
            if (i == D.end())
            {  p = new char[strlen(buf) + 1];
               strcpy(p, buf);
               D[p] = nr;
            }  else (*i).second = nr;
            break;
         case '*':
            for (i = D.begin(); i != D.end(); i++)
               cout  << setw(9) << (*i).second << " "
                     << (*i).first << endl;
            break;
         case '=':
            ofstr.open("phone.txt");
            if (ofstr)
            {  for (i = D.begin(); i != D.end(); i++)
               ofstr    << setw(9) << (*i).second << " "
                        << (*i).first << endl;
               ofstr.close();
            }  else cout << "Cannot open output file.\n";
            break;
         case '#': break;
         default:
            cout << "Use: * (list), ? (find), = (save), "
               "/ (delete), ! (insert), or # (exit).\n";
            cin.getline(buf, maxlen);
            break;
         }
         if (ch == '#') break;
      }
   }

   int main()
   {  directype D;
      ReadInput(D);
      ShowCommands();
      ProcessCommands(D);
      for (directype::iterator i = D.begin(); i != D.end();
         ++i) delete[] (*i).first;
      return 0;
   }
```

Чтобы запустить программу на выполнение, нам не требуется наличие входного файла *phone.txt*. Для нижеследующего примера работы этой программы предположим, что такой файл существует и содержит две записи для абонентов *Smith, P.* и *Johnson, J.* Данные, которые вводит пользователь, подчеркнуты. Заметьте: в полном списке телефонной книги, который вы-

водится с помощью команды *, имена появляются в алфавитном порядке. Помните, что имена являются ключами, хотя и следуют после номеров:

```
Entries read from file phone.txt:
   54321 Smith, P.
   12345 Johnson, J.
Commands:?name       : find phone number,
         /name       : delete
         !number name: insert (or update)
         *           : list whole phonebook
         =           : save in file
         #           : exit
!19723 Shaw, A.
*
   12345 Johnson, J.
   19723 Shaw, A.
   54321 Smith, P.
/Johnson, J.
*
   19723 Shaw, A.
   54321 Smith, P.
?Shaw, A.
Number: 19723
=
#
```

Строки хранятся в памяти, размещаемой с помощью оператора

```
p = new char[strlen(buf) + 1];
```

как при чтении строк из входного файла, так и при вводе пользователем. Освобождение строк с помощью

```
delete[] (*i).first;
```

также происходит в двух случаях: когда пользователь вводит команду удаления и при выходе из программы для освобождения ресурсов. Эти операции не потребуется программировать вручную в версии, которую мы обсудим в разделе 2.12.

Выражение равенства через отношение «меньше»

Имеется еще один интересный аспект, относящийся к использованию С-указателей в качестве ключей. Когда в словаре происходит поиск по ключу, можно ожидать, что это осуществляется с помощью оператора эквивалентности ==. Теперь предположим, что в нашей программе мы ищем в словаре имя "*John*" и существует элемент словаря, содержащий этот ключ

"*John*". Каждый программист на C(++) должен знать, что такой поиск не может быть произведен с помощью оператора ==, поскольку приведет к сравнению адресов, а не самих строк. Вместо == необходимо использовать функцию *strcmp*. Например, при

```
char *s = "John", *t = "John";
```

выражения

```
s == t
strcmp(s, t) == 0
```

будут иметь значения 0 (=*false*) и 1 (=*true*) соответственно.

В нашей программе сравнение на равенство символьных ключей выполнятся правильно, потому что для этой цели используется класс *compare*. Если бы потребовалось сравнить два числа, *a* и *b*, мы могли бы выразить оператор == через <, заменив

```
a == b
```

на

```
!(a < b || b < a)
```

Точно так же проверка на равенство двух ключей *s* и *t* типа *char** осуществляется в нашей программе следующим образом:

```
!(compare()(s, t) || compare()(t, s))
```

Посмотрев на определение функции *operator*() в классе *compare*, мы увидим, что это эквивалентно

```
!(strcmp(s, t) < 0 || strcmp(t, s) < 0)
```

Хотя наше выражение вычисляется медленнее, оно, в свою очередь, эквивалентно

```
strcmp(s, t) == 0
```

2.10. Функции *insert*

Добавление новых записей в предыдущем разделе осуществляется с помощью оператора доступа по индексу в следующем выражении

```
D[p] = nr;
```

Вместо этого оператора присваивания мы могли бы написать выражение

```
D.insert(directype::value_type(p, nr));
```

которое основано на нашем определении

```
typedef map<char*, long, compare> directype;
```

и на следующем определении типа *value_type* в шаблоне *map*, приведенном в заголовке *map*:

```
typedef pair<const Key, T> value_type;
```

где *Key* – тип ключа, а *T* – тип сопутствующих данных. Приведенный выше довольно сложный вызов функции *insert* не имеет преимуществ перед более простым оператором присваивания, если нас не интересует возвращаемое функцией *insert* значение. Это значение может дать нам информацию о результате процесса добавления элемента в словарь. Давайте посмотрим, как определяется функция *insert* в заголовке *map*:

```
pair<iterator, bool> insert(const value_type &x);
```

Очевидно, что *insert* возвращает объект типа *pair*, который содержит итератор, указывающий позицию добавленной (или обновленной) записи и значение типа *bool*, которое показывает, произошло ли добавление новой записи. Это может быть полезным в некоторых приложениях, но в программе *map2.cpp* не имеет значения, потому что, когда пользователь вводит имя, мы начинаем с проверки, является ли это имя новым. Если нет, заменяем только телефонный номер, выполняя

```
(*i).second = nr;
```

без выделения или освобождения памяти. Напомним, что ключи, представляющие имена, являются на самом деле указателями в стиле C.

Рассмотрим теперь другую функцию *insert* для словарей. Если мы знаем позицию, в которой должен быть расположен новый элемент, эта информация может быть использована для того, чтобы операция вставки стала более эффективной по времени выполнения. С этой целью можно использовать функцию *insert*, объявленную как

```
iterator insert(iterator position, const value_type &x);
```

Она возвращает итератор, ссылающийся на элемент, который только что был добавлен в словарь; мы можем использовать это значение, если следующим добавить элемент с большим ключом. Например, если знаем, что элементы будут добавляться по возрастанию ключей, то можем использовать итератор *i*, как это сделано в следующей программе:

```
// mapins.cpp: Быстрая вставка в словарь.
#include <iostream>
#include <map>
using namespace std;
```

```
typedef map<int, double, less<int> > maptype;

int main()
{  maptype S;
   maptype::iterator i = S.end();
   for (int k=1; k<=10; k++)
      i = S.insert(i, maptype::value_type(k, 1.0/k));
   i = S.find(4);
   cout << (*i).second << endl; // Вывод: 0.25
   return 0;
}
```

Первый аргумент этой версии функции *insert* используется как подсказка, указывающая, где следует начинать поиск требуемой позиции. Вставка ускоряется, если новый элемент непосредственно следует за элементом, на который ссылается первый аргумент. В этом случае добавление элемента выполняется за постоянное время. Напротив, если первый аргумент ссылается на любой другой элемент, вставка все равно осуществляется правильно, но занимает время $O(\log N)$, где N – количество элементов множества.

2.11. Удаление элементов словаря

Для удаления элементов словаря имеются три функции-члена *erase*, объявленные следующим образом:

```
void erase(iterator position);
void erase(iterator first, iterator last);
size_type erase(const key_type &x);
```

Первую из этих функций мы использовали в разделе 2.9. Вторая функция дает возможность стереть все элементы заданного диапазона. Третью функцию использовать легче всего, потому что она требует знать значение ключа, а не итератора. Она возвращает 0 или 1, что соответствует числу удаленных элементов. Иными словами, если существует элемент с заданным ключом, функция вернет 1; если нет, она вернет 0.

2.12. Более удобные строки

До сих пор мы использовали достаточно примитивный способ работы со строками. Например, в программе *map.cpp* нам необходимо было выполнить два оператора

```
delete[] (*i).first;
D.erase(i);
```

чтобы удалить элемент множества, на который ссылался итератор *i*. Если мы забудем выполнить первый из этих операторов, это приведет к «утечке

памяти», потому что сама строка символов останется где-то в памяти (по адресу, который хранится в (**i*).*first*) без какой-либо возможности использовать эту память после того, как будет выполнен второй из этих операторов. Способ работы со строками был реализован в духе C, а не C++. Ключами наших элементов словаря являлись указатели на область памяти, которая распределялась и освобождалась вручную.

В проекте стандарта C++ определен библиотечный класс *string,* чтобы упростить манипуляции со строками и уменьшить возможности для совершения случайных ошибок. Класс *string* не является частью STL и сейчас поддерживается не всеми компиляторами. Иными словами, программы, использующие этот класс, на текущий момент не являются переносимыми. Но есть надежда, что ситуация изменится в будущем. Сейчас мы рассмотрим некоторые свойства класса *string*, который поддерживается как BC5, так и VC5.

Чтобы иметь возможность определять переменные класса *string*, необходимо использовать следующую строку *#include:*

```
#include <string>
```

и на этот раз мы не вправе написать <*string.h*> вместо <*string*>. Если ваш компилятор не принимает этот заголовок, он может поддерживать заголовок *bstring.h* или *cstring.h*, который вы должны использовать в таком случае. Иногда эти заголовки нужно включать *перед* заголовками STL. После этого мы можем написать следующие операторы присваивания:

```
string s, t ;
s = "ABC";
t = "DEFGH" ;
s = t;      // s = "DEFGH"
s = 'A';    // s = "A"
```

Как показывает приведенный фрагмент, с правой стороны этих присваиваний может находиться значение одного из трех типов:

- *string* (как в *s = t;*);
- *char** (как в *s = «ABC»;*);
- *char* (как в *s = «A»;*).

То же относится и к оператору +=, применяющемуся для добавления второго операнда к концу первого, который должен принадлежать к типу *string*:

```
string s, t("KLM"), u(t);
                // s = "", t = u = "KLM"
s += t;         // s = "KLM"
s += "PQR";     // s = "KLMPQR"
```

```
s += 'A';     // s = "KLMPQRW"
```

Этот фрагмент, кроме того, показывает, что мы можем инициализировать строку с помощью значения типа *char** либо *string*.

Существуют два других способа инициализации строк. Первый – указать повторитель; например, начальное значение *s* будет «*AAAA*», если мы напишем

```
string s(5, 'A');
```

Второй способ – использовать два итератора, например:

```
string t(v.begin(), v.end()), u(a, a+3);
```

где *v* имеет тип *vector<char>*, а *a* является массивом символов, первые три элемента которого используются для инициализации *u*.

Класс *string* имеет функцию-член *c_str*(), возвращающую указатель типа *const char**, который можно применить в выражениях со строками C. Например, после

```
string s("AB");
char a[10];
strcpy(a, s.c_str());
```

мы имеем

```
a[0] = 'A'
a[1] = 'B'
a[2] = '\0'
```

Как в случае с вектором, мы можем получить текущий размер строки с помощью функции-члена *size*. Например, после определения *s* в вышеприведенном фрагменте имеем

```
s.size() = 2
```

Есть много интересных способов работы со строками, некоторые из них используются в следующей программе:

```
// strdemo.cpp: Программа, демонстрирующая возможности
//              класса string.

#include <iostream>
#include <string>

using namespace std;

int main()
{  string s(5, 'A'), t("BC"), u;
   u = s + t;
   cout << "u = s + t = " << u << endl;
```

```
   cout << "u.size() = " << u.size() << endl;
   cout << "u[6] = " << u[6] << endl;
   cout << "Enter two new strings s and t:\n";
   cin >> s >> t;
   cout << "s.size() = " << s.size() << endl;
   cout << (s < t ? "s < t" : "s >= t");
   cout << endl;
   return 0;
}
```

Эта программа показывает, что для соединения двух строк можно использовать оператор + (как и обсужденный выше оператор +=). Для ввода и вывода определены операторы >> и <<, и ведут они себя так же, как и для обычных строк. Это справедливо и для оператора доступа по индексу [], как показано при выводе значения *u*[6]. И наконец, для сравнения строк мы можем применить операторы ==, !=, <, >, <= и >=, как показано на примере оператора <. Ниже дан пример работы этой программы, в котором подчеркнут текст, введенный пользователем:

```
u = s + t = AAAAABC
u.size() = 7
u[6] = C
Enter two new strings s and t:
xxxxx xxy
s.size() = 5
s < t
```

Несмотря на различные возможные длины строк, экземпляры класса *string* могут быть элементами массивов и контейнеров STL, например:

```
// strelem.cpp: Строки C++ в качестве элементов
//    списка L и массива a.
#include <iostream>
#include <string>
#include <list>
#include <algorithm>
using namespace std;

int main()
{  string s("One"), t("Two"), u("Three");
   list<string> L;
   L.push_back(s);
   L.push_back(t);
   L.push_back(u);
   string a[3];
   copy(L.begin(), L.end(), a);
   for (int k=0; k<3; k++) cout << a[k] << endl;
```

```
   return 0;
}
```

Как легко предсказать, вывод этой программы будет следующий:

```
One
Two
Three
```

Теперь давайте займемся применением класса *string* для второго способа решения задачи, рассмотренной в разделе 2.9. Вспомним, что программа *map2.cpp* сперва пыталась прочесть файл *phone.txt*, чтобы использовать его в качестве телефонного справочника. Если такого файла не существует, с помощью данной программы его можно создать. Программа *map2a.cpp* отличается тем, что она не использует операторы *new* и *delete* для строк в стиле C, а вместо этого – новый тип *string*. Комментарии, начинающиеся с // !, указывают, чем эта программа отличается от *map2.cpp*:

```
// map2a.cpp: Вторая версия телефонного справочника,
//            использующая тип 'string'.
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <map>
using namespace std;
                                                    // !
typedef map<string, long, less<string> > directype;

// Прочитать все символы до '\n' и сохранить их в str,
// за исключением '\n', который будет прочитан, но не сохранен.
void getaline(istream &is, string &str)
{  char ch;
   str = "";                                        // !
   for (;;)
   {  is.get(ch);
      if (is.fail() || ch == '\n') break;
      str += ch;
   }
}

void ReadInput(directype &D)
{  ifstream ifstr("phone.txt");
   long nr;
   string str;                                      // !
   if (ifstr)
   {  cout << "Entries read from file phone.txt:\n";
      for (;;)
```

```
      {  ifstr >> nr;
         ifstr.get(); // пропустить пробел
         getaline(ifstr, str);                    // !
         if (!ifstr) break;
         cout << setw(9) << nr << " " << str << endl;
         D[str] = nr;                             // !
      }
   }
   ifstr.close();
}

void ShowCommands()
{  cout <<
      "Commands: ?name       : find phone number,\n"
      "          /name       : delete\n"
      "          !number name: insert (or update)\n"
      "          *           : list whole phonebook\n"
      "          =           : save in file\n"
      "          #           : exit" << endl;
}

void ProcessCommands(directype &D)
{  ofstream ofstr;
   long nr;
   string str;                                    // !
   char ch;
   directype::iterator i;
   for (;;)
   {  cin >> ch;  // пропустить пустые символы
                  // и прочитать ch
      switch (ch){
      case '?': case '/': // найти или удалить:
         getaline(cin, str);
         i = D.find(str);                        // !
         if (i == D.end())
            cout << "Not found.\n";
         else              // Ключ найден.
         if (ch == '?')    // Команда 'Найти'
            cout << "Number: " << (*i).second << endl;
         else              // Команда 'Удалить'
            D.erase(i);                          // !
         break;

      case '!':            // добавить (или обновить)
         cin >> nr;
         if (cin.fail())
         {  cout << "Usage: !number name\n";
            cin.clear();
            getaline(cin, str);                  // !
```

```
            break;
         }
         cin.get();        // пропустить пробел;
         getaline(cin, str);
         D[str] = nr;                              // !
         break;
      case '*':
         for (i = D.begin(); i != D.end(); i++)
            cout  << setw(9) << (*i).second << " "
                  << (*i).first << endl;
         break;
      case '=':
         ofstr.open("phone.txt");
         if (ofstr)
         {  for (i = D.begin(); i != D.end(); i++)
               ofstr << setw(9) << (*i).second << " "
                     << (*i).first << endl;
            ofstr.close();
         }  else cout << "Cannot open output file.\n";
         break;
      case '#': break;
      default:
         cout << "Use: * (list), ? (find), = (save), "
            "/ (delete), ! (insert), or # (exit).\n";
         getaline(cin, str); break;                // !
      }
      if (ch == '#') break;
   }
}

int main()
{  directype D;
   ReadInput(D);
   ShowCommands();
   ProcessCommands(D);
   return 0;                                       // !
}
```

Поскольку поведение этой программы не отличается от поведения начальной версии, *map2.cpp*, опустим пример ее работы. Эта версия проще в том отношении, что мы оперируем с объектами класса *string* таким же образом, как со встроенными типами, например *int*, несмотря на различные длины строк. Так как память для этих строк размещается и освобождается автоматически, в программе не встречаются операторы *new* и *delete*.

3

Последовательные контейнеры

3.1. Векторы и связанные с ними типы

Когда идет речь о шаблонах вообще и STL в частности, новички бывают сбиты с толку сложным синтаксисом этих выражений. Мы начнем с обсуждения синтаксиса, приводя примеры, которые не всегда имеют практическое значение, но полезны тем, что позволяют в нем разобраться.

Во второй главе книги мы использовали тип *value_type* при работе с ассоциативными контейнерами. Он определен также для последовательных контейнеров. В классе *vector* указанный тип определен достаточно простым способом:

```
typedef T value_type;
```

в то время как в классе *map* определение этого типа следующее:

```
typedef pair<const Key, T> value_type;
```

В любом случае мы видим, что *value_type* означает тип элементов, содержащихся в контейнере. Поскольку его определение находится внутри рассматриваемого класса, при использовании в другом контексте мы должны квалифицировать имя *value_type*, например таким образом:

```
vector<int>::value_type;
```

Из предыдущего обсуждения следует, что эта усложненная запись обозначает просто тип *int*. Другими словами, если мы используем заголовок *vector*, то можем заменить строчку

```
int k;
```

на

```
vector<int>::value_type k;
```

Вряд ли эта форма более удобна, но отсюда видно, что запись *vector<int>::value_type* не содержит ничего мистического.

Перечисленные ниже типы могут быть использованы точно таким же образом, как типы, определенные одним идентификатором:

```
vector<int>::iterator
vector<int>::reverse_iterator
vector<int>::const_iterator
vector<int>::const_reverse_iterator
```

Вспомним, что мы рассматривали первые три из этих четырех типов в разделах 1.2 и 1.3. Четвертый, очевидно, может быть использован для прохождения контейнера в обратном порядке без изменения его элементов. Например, после

```
typedef vector<int>::iterator VecIntIterType;
```

мы можем заменить

```
vector<int>::iterator i;
```

на более простое выражение

```
VecIntIterType i;
```

Вот полный список вложенных типов:

```
value_type
reference, const_reference
iterator, const_iterator
reverse_iterator, const_reverse_iterator
difference_type
size_type
vector_allocator
```

Их имена хорошо объясняют их назначение. Нам потребуется большинство из этих типов при обсуждении функций-членов контейнеров. Далее рассмотрим два оператора доступа по индексу для векторов, которые могут быть объявлены в шаблоне *vector* следующим образом:

```
reference operator[](size_type n);
const_reference operator[](size_type n) const;
```

Если мы имеем дело с типом *vector<int>*, тип *reference* означает *int&*, или *ссылку-на-целое*. Тип *difference_type* представляет собой целочисленное значение со знаком, которое может использоваться для представления разности между двумя итераторами, а *size_type* отличается от *difference_type* только тем, что является беззнаковым типом.

Тип *vector<bool>*

Векторы, состоящие из элементов типа *bool*, представляют собой особый случай, чтобы обеспечить их эффективное размещение: выделение целого слова на каждый элемент, которому достаточно одного бита, было бы неэкономным расходованием памяти. Очевидно, особый подход к типу *bool* возможен только тогда, когда он является встроенным типом, а не синонимом типа *int*. Напомним, что эта тема затрагивалась нами в разделе 1.1.

Функция-член *flip*() определена для типа *vector<bool>*, но не для *vector<int>*. Она обращает значение всех либо выбранного бита в векторе, как показывает следующий пример:

```
vector<bool> b(100); // 100 бит, все установлены в 0.
b.flip(); // Все биты инвертированы и сейчас равны 1.
b[73].flip();
cout << b[72] << b[73] << b[74]; // Вывод: 101
```

Также существует функция-член *swap* для перестановки двух булевских векторов. Например:

```
vector<bool> u(100, true), v(50, false);
u.swap(v);
cout  << u.size() << " "
      << v.size() << " "
      << u[0];                    // Вывод: 50 100 0
```

Следующая программа использует булевский вектор для реализации *решета Эратосфена*. Это хорошо известный эффективный способ нахождения простых чисел 2, 3, 5, 7, 11, 13, ... Все элементы булевского вектора *S*, решета, сначала равны *true*, а впоследствии некоторые из них принимают значение *false*:

S[*i*] = *true* означает:	i может быть простым числом (делители пока не найдены);
S[*i*] = *false* означает:	i не есть простое число (поскольку является кратным меньшего простого числа).

Чтобы уменьшить объем выводимых данных, программа не показывает длинный список простых чисел. Вместо этого она подсчитывает количество простых чисел, меньших заданного значения *N*, а также печатает наибольшее из них:

```
// erastos.cpp: Решето Эратосфена для нахождения
//                  всех простых чисел, меньших заданного
предела.

#include <iostream>
#include <vector>
#include <math.h>

using namespace std;

int main()
{  cout <<
      "To generate all prime numbers < N, enter N: ";
   long N, i, sqrtN, count = 0, j;
   cin >> N;
   sqrtN = int(sqrt(N)) + 1;
   vector<bool> S(N, true);

   // Вначале все S[i] равны true.
   // S[i] = false только и когда
   // мы обнаруживаем, что i не есть простое.

   for (i=2; i < sqrtN; i++)
      if (S[i])
         for (int j=i*i; j<N; j+=i) S[j] = false;

   for (i=2; i<N; i++)
      if (S[i]) {j = i; count++;}

   cout  << "There are " << count
         << " prime numbers less than N.\n";
   cout  << "Largest prime number less than N is "
         << j << "." << endl;
   return 0;
}
```

Ниже следует пример результатов работы этой программы, который показывает, что количество найденных простых чисел действительно может быть слишком большим для того, чтобы показать их все на экране:

```
To generate all prime numbers < N, enter N: 1000000
There are 78498 prime numbers less than N.
Largest prime number less than N is 999983.
```

Время, которое потребовалось на выполнение примера, составило около 10 секунд на компьютере с процессором 80486; это показывает, что *решето Эратосфена* позволяет очень быстро находить простые числа. Также этот пример показывает, что булевские векторы не только очень удобны, но и эффективны.

3.2. Функции *capacity* и *reserve*

До сих пор мы принимали как данность, что векторы имеют переменный размер, не беспокоясь о том, как это реализовано. Теперь предположим, что мы хотим добавить к вектору элемент, так что размер этого вектора вырастет на 1. Если такое будет происходить часто, то неэффективно было бы каждый раз перераспределять память, потому что это может потребовать копирования всех элементов в область памяти, в которой размещается увеличенный вектор. Гораздо лучше выделить больше памяти, чем требуется, так что во многих случаях дополнительная память уже будет иметься в наличии, когда возникнет необходимость в увеличении размера вектора. Число элементов вектора *v*, для которых выделена память, называется *емкостью* вектора и равняется значению выражения

```
v.capacity()
```

Это значение больше или равно значению *v.size*(), как показано на рисунке 3.1.

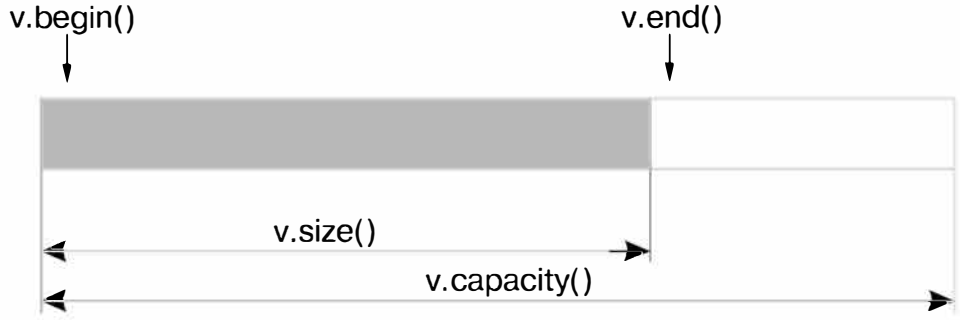

Рисунок 3.1. Размер и емкость вектора **v**

Следующая программа демонстрирует, что значение *v.capacity*() остается постоянным относительно долго, пока вектор *v* постепенно растет, а *v.size*() увеличивается на единицу каждый раз, когда к *v* добавляется новый элемент:

```
// capacity.cpp: Функция-член capacity
//               для векторов.

#include <iostream>
#include <iomanip>
#include <vector>
```

```
using namespace std;

int main()
{  vector<int> v;
   vector<int>::size_type n0 = 12345, n1;
   cout << "v.size()  v.capacity()\n";

   for (long i=0L; i<100000L; i++)
   {  n1 = v.capacity();
      if (n1 != n0)
      {  cout  << setw(8) << v.size() << "      "
               << setw(8) << n1 << endl;
         n0 = n1;
      }
      v.push_back(123); // v.size() увеличивается на 1
   }
   return 0;
}
```

В этой программе размер вектора *v* вырастает от 0 до 99 999. Число элементов вектора, для которых выделена память, *v.capacity*() показывается каждый раз, когда это значение изменяется по сравнению с прошлым шагом. Значения, которые хранятся в векторе, не имеют отношения к нашей текущей задаче, все элементы вектора равны 123. В начале работы программы вектор пуст и значение *v.size*(), как и *v.capacity*(), равно нулю. После добавления первого элемента значение *v.size*() становится равным 1, но *v.capacity*(), количество элементов, для которых зарезервирована память, равно некоторому большему значению. Для BC5 это значение 256, для некоторых других версий STL оно равно 1024. При, казалось бы, неэффективном расходовании памяти такая схема имеет преимущество в том, что потребуется много времени, прежде чем возникнет необходимость в перераспределении памяти, поскольку емкость, равная, допустим, 256, используется для всех размеров вектора 1, 2, 3, ..., 256. Только когда значение *v.size*() станет равным 257, нам потребуется перераспределить память. Для используемой здесь реализации STL емкость после этого удвоится, чтобы хватило памяти для размеров 257, 258, ..., 512. Принцип удвоения емкости соблюдается при всех перераспределениях памяти, как показывает следующий вывод программы, откомпилированной с помощью BC5:

```
v.size()  v.capacity()
       0             0
       1           256
     257           512
     513          1024
    1025          2048
```

```
 2049          4096
 4097          8192
 8193         16384
16385         32768
32769         65536
65537        131072
```

Функция *capacity* возвращает информацию о распределении памяти; существует связанная с ней функция *reserve*, которая позволяет контролировать это распределение. После выполнения вызова

```
v.reserve(n);
```

значение *v.capacity*() будет равно по меньшей мере *n*. Функция *reserve* может ускорить выполнение программы, если мы знаем заранее, сколько элементов будет содержать вектор *v*. Например, если в программе *capacity.cpp* добавить строчку

```
v.reserve(100000);
```

непосредственно перед циклом *for*, вывод этой программы будет содержать только две следующие строчки:

```
v.size()  v.capacity()
       0        100000
```

Выделение памяти произойдет всего один раз. После этого вызова *reserve* в векторе хватит места для всех 100 000 элементов, которые будут добавлены в операторе *for*.

Перераспределение памяти, значения итераторов и функция *reserve*

Когда в результате роста происходит перераспределение выделенной памяти для вектора *v*, что приводит к увеличению значения *v.capacity*(), то, как правило, любые итераторы, ссылающиеся на элементы вектора *v*, становятся недействительными. Это станет ясно, если считать итераторы указателями, которые хранят адреса элементов. Перераспределение памяти может потребовать, что весь вектор будет перемещен в другие участки памяти, и мы не можем ожидать, что использовавшиеся с этим вектором итераторы автоматически обновятся. Например:

```
vector<int> v, w;
...
vector<int>::iterator i;
v.push_back(0);
i = v.begin();
for (long k=1L; k<100000L; k++)
```

```
    v.push_back(k);
cout << (*i);  // ???
```

В последней строке этого фрагмента *i* ссылается на участок памяти, который может уже не принадлежать вектору *v*. Как если бы мы захотели навестить старого знакомого по адресу, где он когда-то проживал, но сейчас там живут другие люди. Однако есть исключения; если вторая строчка этого фрагмента (обозначенная тремя точками) содержит оператор

```
v.reserve(N);
```

где $N \geqslant 100\,000$, тогда в цикле *for* не произойдет перераспределения памяти и значение *i* останется равным *v.begin*().

Как видно из раздела 1.9, мы можем использовать выражения $i + n$ и $i - n$, где *i* является итератором, а *n* – целым, для векторов (а также для двусторонних очередей, но не для списков). Это дает возможность правильно запомнить позиции в векторе:

```
vector<int> v;
vector<int>::iterator i;
int k;
for (...) v.push_back(...);
i = ...;
k = i - v.begin();          // *i == v[k]
for (...) v.push_back(...);
// Расширение v может привести к перераспределению памяти,
// что сделает *i неопределенным.
i = v.begin() + k;          // *i == v[k] снова
// Итератор i ссылается на тот же элемент, что и до этого,
// хотя этот элемент, возможно, находится в другом месте
// памяти
```

Как отмечается в комментарии, нам нет необходимости использовать итераторы для доступа к элементам вектора: оператор доступа по индексу [], который применяется к массивам, определен также и для векторов, поэтому в конце этого фрагмента мы можем написать *v*[*k*] вместо **i*. В общем случае можно писать

```
v[k]  вместо  *(v.begin() + k)
```

Функция-член вектора *erase*, рассмотренная в разделе 1.3, также делает недействительными все итераторы, ссылающиеся на элементы вектора, расположенные после удаляемого, поскольку эти элементы будут передвинуты, чтобы заполнить пробел, возникший при выполнении операции *erase*.

3.3. Обзор функций-членов класса *vector*

Ниже перечислены все функции-члены класса *vector* с кратким описанием или ссылкой на соответствующий раздел. Эти объявления могут содержаться в заголовке *vector*, хотя многие функции-члены полностью *определены*, а не только *объявлены* в этом заголовке.

```
iterator begin();
iterator end();
void push_back(const T& x);
reverse_iterator rbegin();
reverse_iterator rend();
```

Рассмотрены в разделе 1.2.

```
const_iterator begin() const;
const_iterator end() const;
```

Рассмотрены в разделе 1.3 на примере списков.

```
const_reverse_iterator rbegin() const;
const_reverse_iterator rend() const;
```

Для прохождения в обратном порядке в режиме «только для чтения».

```
size_type size() const;
size_type capacity() const;
void reserve(size_type n);
```

Рассмотрены в разделе 3.2.

```
size_type max_size() const;
```

Очень большое целое число (порядка 1 073 741 823), показывающее, до каких размеров может вырасти вектор.

```
bool empty() const;
```

Позволяет определить, пуст ли вектор.

```
vector();                                              // (1)
vector(size_type n, const T& value = T());             // (2)
vector(const vector<T>& x);                            // (3)
vector(const_iterator first, const_iterator last);     // (4)
~vector();
```

Четыре конструктора и деструктор. Эти конструкторы используются в следующих примерах:

```
vector<int> v;            // (1) Конструктор по умолчанию
```

```
vector<int> w(5, -3);      // (2) Создает 5 элементов,
                           //     все они равны -3
vector<int> w(5);
vector<int> w1(w);         // (3) Создает 5 элементов
vector<int> w2(w.begin()+1, w.begin() + 5);
                           // (4) Копирует четыре элемента
                           //     из w в w2
```

```
reference operator[](size_type n);
const_reference operator[](size_type n) const;
```

Операторы доступа по индексу используются в

```
w[3] = 3 * w[2] + 1;
```

```
vector<T>& operator=(const vector<T>& x);
```

Оператор присваивания используется в

```
vector<int> w(5, -3), v;
v = w;
```

```
reference front();
reference back();
const_reference front() const;
const_reference back() const;
```

Эти функции обеспечивают доступ к первому и последнему элементам вектора. Например, можно написать:

```
vector<int> v;
for (int i=10; i<15; i++) v.push_back(i);
v.front() = 1000;
cout << v.front() << " " << v.back() << " ";
cout << "Size = " << v.size() << endl;
// Вывод: 1000 14 Size = 5
```

```
void swap(vector<T>& x);
```

Мы можем написать либо

```
v.swap(w);
```

или

```
w.swap(v);
```

чтобы поменять местами векторы v и w одного типа (но необязательно одного размера).

```
iterator insert(iterator position, const T& x);
```

Служит для вставки элемента в заданной позиции. Эта и следующие функции *insert* занимают время $O(n)$, где n – количество элементов, на-

ходящихся после вставляемого элемента; напомним, что эти элементы должны быть передвинуты, чтобы освободить место для вставляемых. Функция возвращает итератор, который ссылается на вставленный элемент. Например, фрагмент кода

```
vector<int> v;
for (int k=10; k<15; k++) v.push_back(k);
vector<int>::iterator i = v.begin() + 1, j;
j = v.insert(i, 123);
copy(v.begin(), v.end(),     // См. раздел 1.9.
   ostream_iterator<int>(cout, " ")); cout <<endl;
cout << "j refers to " << *j << endl;
```

дает следующий вывод:

```
10 123 11 12 13 14
j refers to 123
```

```
void insert(iterator position, const_iterator first,
   const_iterator last);
```

позволяет нам вставить более одного элемента за раз. Вставляемые элементы могут быть заданы любым действительным диапазоном [*first, last*) элементов требуемого типа. Вот пример:

```
vector<int> v;
for (int k=0; k<5; k++) v.push_back(k);
int a[3] = {100, 200, 300};
v.insert(v.begin() + 1, a, a+3);
// Содержимое v: 0 100 200 300 1 2 3 4
```

```
void insert (iterator position, size_type n, const T& x);
```

Эта функция служит для вставки нескольких одинаковых элементов, равных третьему аргументу, *x*. Например:

```
vector<int> v;
for (int k=0; k<5; k++) v.push_back(k);
int initvalue = -1;
v.insert(v.begin() + 1, 4, initvalue);
// Содержимое v: 0 -1 -1 -1 -1 1 2 3 4
```

```
void pop_back();
void erase(iterator position);
```

См. раздел 1.3.

```
void erase(iterator first, iterator last);
```

Удаляются элементы диапазона [*first, last*), как показывает следующий пример:

```
vector<int> v;
for (int k=0; k<7; k++) v.push_back(10 * k);
   // Содержимое v: 0, 10, 20, 30, 40, 50, 60
v.erase(v.begin()+2, v.begin()+5);
   // Содержимое v: 0, 10, 50, 60
```

Обе функции *erase* занимают время $O(n)$, где n – число элементов, которые следуют за удаляемыми элементами. Если нам нужно удалить несколько последовательных элементов вектора, мы должны использовать один вызов последней функции *erase*, поскольку это значительно быстрее, чем несколько раз вызывать предпоследнюю функцию, которая удаляет по одному элементу за раз.

3.4. Двусторонние очереди

Различные типы контейнеров очень похожи в отношении способов их применения. Например, каждый из трех типов контейнеров (*vector*, *deque* и *list*) определяет конструктор, который использует в качестве параметров число повторений и значение, например:

```
vector<double> v(100, 12.34);
deque<double> D(100, 12.34);
list<double> L(100, 12.34);
```

Каждое из этих объявлений создает последовательность, состоящую из 100 экземпляров значения 12.34. Из примера следует, что можно сократить наше обсуждение двусторонних очередей и списков, ограничившись рассмотрением специфических свойств этих контейнеров и делая ссылку на векторы относительно общих аспектов.

Наше обсуждение двусторонних очередей в разделах 1.3 и 1.9 позволяло сделать вывод, что эти структуры данных обладают только преимуществами перед векторами. Кроме функций-членов, определенных для векторов, они определяют функции-члены *push_front* и *pop_front*. Так как одна из существенных характеристик двусторонней очереди – это возможность расти и сокращаться с двух сторон, кажется естественным реализовать двустороннюю очередь с помощью двусвязного списка. Однако в STL двусторонняя очередь реализована по-другому, поскольку, что удивительно, двусторонняя очередь в STL обеспечивает произвольный доступ к элементам за постоянное время. На чем основана такая реализация? Рисунок 3.2 дает набросок модели реализации. Элементы двусторонней очереди размещаются в блоках фиксированного размера. Кроме этого,

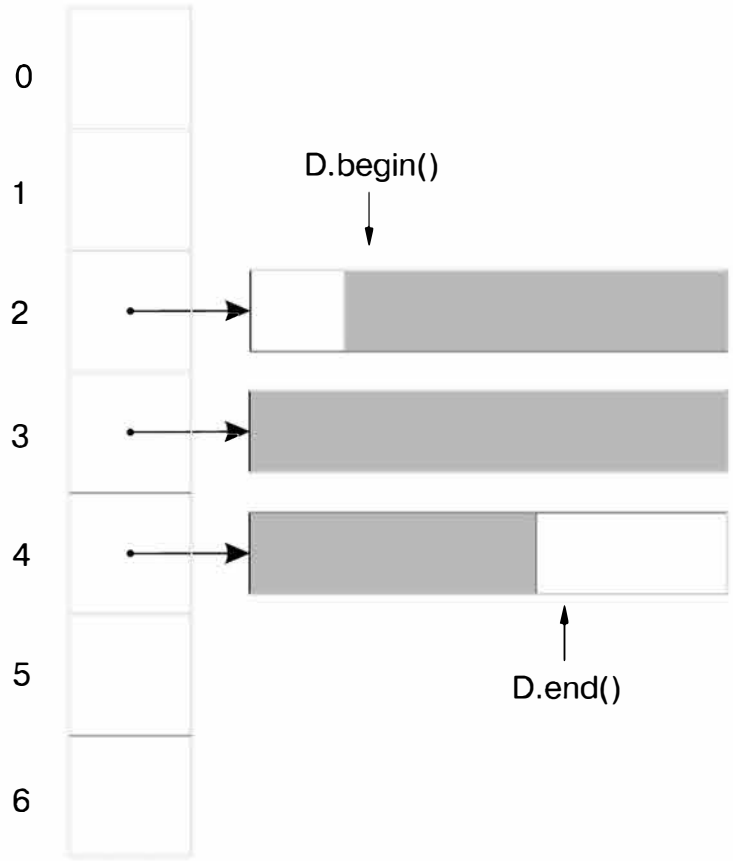

Рисунок 3.2. Возможная реализация двусторонней очереди **D**

хранится массив указателей на эти блоки. Заштрихованные области блоков действительно используются контейнером, а белые области означают выделенную, но неиспользуемую память.

Выполнение

```
D.push_front(x);
```

предполагает, что элемент x помещается слева от позиции, обозначенной *D.begin*(). Эта вставка в начале D заставляет первую стрелку ↓ сместиться влево. После того как эта операция выполнится несколько раз, первый из блоков, изображенных на рисунке, станет заполненным. Тогда будет размещен новый блок, указатель на который будет помещен в позицию 1 массива, изображенного слева. Когда и этот блок заполнится, будет выделен следующий блок, указатель на который займет место в позиции 0 массива. Может показаться, что должна возникнуть проблема, когда после нескольких дополнительных вставок и этот блок также заполнится. Однако с этой проблемой можно справиться с достаточной эффективностью, перераспределив память для массива указателей. Когда у нас появятся пять блоков, мы выделим память под больший массив указателей, в котором будут использоваться только пять элементов в середине, что даст

возможность снова расширять структуру с обоих концов. Легко видеть, что вставка в конце (а не в начале) работает по сходным механизмам.

Не существует гарантий, что после вставки элементов итераторы, которые ссылаются на двусторонние очереди, останутся действительными. Напомним, что это утверждение справедливо и для векторов, как мы видели в разделе 3.2. Так как все блоки на рисунке 3.2 обладают одинаковой длиной, адрес элемента двусторонней очереди может быть вычислен за постоянное время; другими словами, двусторонние очереди предоставляют произвольный доступ к своим элементам. Мы даже можем использовать применяемую для массивов запись

```
D[k]  вместо  *(D.begin() + k)
```

Учитывая более сложную схему распределения памяти, можно ожидать, что операции с двусторонними очередями будут выполняться несколько медленнее, чем операции с векторами. Это объясняет логику, по которой в STL имеются обе эти структуры: функции *push_front* и *pop_front,* выполняющиеся за постоянное время, доступны для двусторонних очередей, но не для векторов, но большинство тех операций, которые определены для обоих контейнеров, вероятно, будут выполняться несколько быстрее для векторов по сравнению с двусторонними очередями.

Для двусторонних очередей не определены функции *reserve* и *capacity* (см. раздел 3.2), но есть функция-член *size*, и, как обычно,

```
D.size() эквивалентно  D.end - D.begin()
```

Большинство функций двусторонних очередей ведут себя так же, как их аналоги для векторов (см. раздел 3.3). Вот несколько объявлений функций-членов класса *deque*, которые отсутствуют в классе *vector*:

```
void push_front(const T& x);
void pop_front();
reference front();
```

Например, предположим, у нас имеются вектор v и двусторонняя очередь D, оба непустые и хранящие элементы одного и того же типа. Тогда следующий фрагмент заменяет последний элемент v первым элементом D, после чего первый элемент D стирается:

```
v.back() = D.front();
D.pop_front();
```

3.5. Списки

Как известно, преимущество списков перед векторами и двусторонними очередями заключается в том, что вставка и удаление элементов в любой позиции происходит за постоянное время, а недостаток заключается в отсутствии произвольного доступа к элементам. Список STL может быть просто реализован в виде двусвязного списка, как показано на рисунке 3.3. Поскольку каждый узел этого списка содержит ссылку на предыдущий и последующий узлы, операции –– и ++ для итераторов будут выполняться эффективно, то есть за постоянное время. Напротив, передвижение от узла к узлу, отстоящему на n позиций в списке, потребует времени $O(n)$. Чтобы вставить узел в середине списка, требуется присвоить новые значения четырем указателям, содержащимся в узлах, два из них будут указывать на этот узел (от правого и левого соседа), и два других из самого узла – на его соседей. Остальные узлы останутся незатронутыми, отсюда и следует, что вставка в любой позиции происходит за постоянное время. Удаление узла происходит с той же эффективностью.

Если узлы добавляются в список, все итераторы, ссылающиеся на узлы этого списка, продолжают оставаться действительными; как мы знаем, для векторов и двусторонних очередей это правило не соблюдается.

Многие функции-члены, такие как конструкторы и функции вставки, используются так же, как и функции векторов и двусторонних очередей, но, как мы только что упоминали, для списка добавление элементов занимает постоянное время. У списка отсутствует оператор доступа по индексу [], ведь доступ по индексу связан с итераторами произвольного доступа, а итераторы списка только двунаправленные.

Функции-члены *sort* и *unique*

В разделе 1.4 было сказано, что алгоритм *sort* не работает со списками. Вместо этого для списков определена функция-член *sort*. Другой специфичной для списка функцией-членом является *unique*, которая удаляет повторяющиеся последовательные элементы. Эти две функции определены в классе *list* таким образом:

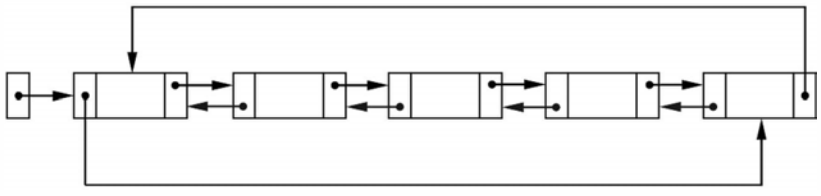

Рисунок 3.3. Двусвязный список

```
void sort();
void unique();
```

Следующая программа демонстрирует использование этих функций:

```
// list1.cpp: Функции-члены класса list: sort и unique.
#include <iostream>
#include <list>
using namespace std;

void out(char *s, const list<int> &L)
{  cout << s;
   copy(L.begin(), L.end(),
   ostream_iterator<int>(cout, " "));
   cout << endl;
}

int main()
{  list<int> L(5, 123);
   L.push_back(100);
   L.push_back(123);
   L.push_back(123);
   out("Initial contents: ", L);
   L.unique();
   out("After L.unique(): ", L);
   L.sort();
   out("After L.sort():   ", L);
   return 0;
}
```

Вывод этой программы показывает, что функция *unique* учитывает только последовательные элементы, в результате получается два элемента 123: один до и другой после элемента 100.

```
Initial contents: 123 123 123 123 123 100 123 123
After L.unique(): 123 100 123
After L.sort():   100 123 123
```

Если бы мы вызвали две функции-члена в другом порядке

```
L.sort(); L.unique();
```

окончательный результат содержал бы всего два элемента

```
100 123
```

Сцепка

Другая операция, специфичная для списков,– сцепка (splicing), перемещение одного или более последовательных элементов из одного списка в другой, без освобождения либо выделения памяти для этих элементов. Существуют три функции-члена *splice*, которые объявлены в классе *list* так,

как показано ниже. В сопутствующем обсуждении *L* и *M* означают список одного и того же типа:

```
void splice(iterator position, list<T>& x);
```

Если *i* является допустимым итератором для *L*, следующая операция вставляет содержимое *M* перед *i* в список *L* и оставляет *M* пустым. Эта операция *не* работает, если *L* и *M* являются одним и тем же списком:

```
L.splice(i, M);
```

```
void splice(iterator position, list<T>& x, iterator j);
```

Если *i* является допустимым итератором для *L*, а *j* – таковым для *M*, следующая операция удаляет элемент, на который ссылается *j*, и вставляет его перед *i*. Эта операция работает, даже если *L* и *M* являются одним и тем же списком:

```
L.splice(i, M, j);
```

```
void splice(iterator position, list<T>& x,
     iterator first, iterator last);
```

Если *i* является допустимым итератором для *L*, а [*j*1, *j*2) является допустимым диапазоном для *M*, следующая операция удаляет элементы этого диапазона и вставляет их перед *i* в *L*. Эта операция также работает, если *L* и *M* являются одним и тем же списком:

```
L.splice(i, M, j1, j2);
```

Мы покажем, как работает последняя из функций *splice*, изменив список так, как изображено на рисунке 3.4.
Элементы 20, 30, 40 из последовательности 10, 20, 30, 40, 50, 60, 70, 80 перемещаются на новое место между 60 и 70. Результатом является последовательность

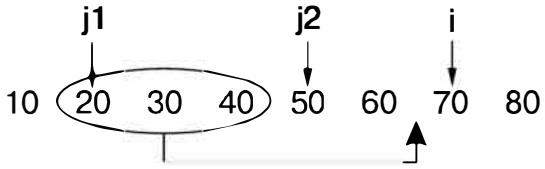

Рисунок 3.4. Сцепка

```
10 50 60 20 30 40 70 80
```

которая совпадает с выводом нижеследующей программы:

```
// splice.cpp: Сцепка.
#include <iostream>
#include <list>
using namespace std;
```

```
int main()
{  list<int> L;
   list<int>::iterator i, j1, j2, j;
   for (int k = 10; k <= 80; k += 10)
   {  L.push_back(k);
      j = L.end();
      if (k == 20) j1 = --j; else
      if (k == 50) j2 = --j; else
      if (k == 70) i = --j;
   }

   L.splice(i, L, j1, j2);
   copy(L.begin(), L.end(),
      ostream_iterator<int>(cout, " "));
   cout << endl;
   return 0;
}
```

Обратим внимание на оператор

```
i = --j;
```

и подобные ему в этой программе. Мы не можем заменить его на

```
i = j - 1;
```

потому что операторы «плюс» и «минус» не определены для двунаправленных итераторов (см. раздел 1.9). Здесь требуется использовать оператор `--`, так как итератор *j* = *L.end*() ссылается на позицию после последнего элемента, который был добавлен с помощью *push_back*. Вместо решения с тремя операторами *if* мы могли бы использовать алгоритм *find*, заменив довольно сложный цикл *for*, приведенный выше, на следующий фрагмент:

```
for (int k = 10; k <= 80; k += 10)
   L.push_back(k);
j1 = find(L.begin(), L.end(), 20);
j2 = find(L.begin(), L.end(), 50);
i = find(L.begin(), L.end(), 70);
```

Данный фрагмент вполне приемлем для короткой последовательности, используемой в этой программе, но решение, приведенное в полном примере, более эффективно для очень длинных последовательностей.

Стоит вспомнить, что алгоритмы, такие как *find*, объявлены в заголовке *algorithm*, поэтому мы могли бы написать

```
#include <algorithm>
```

в начале нашей программы. Для BC5 этот заголовок включается косвенным образом такими заголовками, как *vector* или *list*, поэтому мы опустили эту строчку, полагая, что другие современные версии STL ведут себя аналогично.

Функция-член *remove* класса *list*

Когда нам требуется удалить все элементы списка с заданным значением, решение в два этапа, использующее алгоритмы *remove* и *erase*, рассмотренное в разделе 1.13, является не самым эффективным. Гораздо лучше для этой цели подойдет функция-член *remove*, определенная для списков. Она объявлена в шаблонном классе *list* следующим образом:

```
void remove(const T& value);
```

Эта функция специфична для списков; она не определена как член для векторов и двусторонних очередей. Следующая программа показывает, что *remove* довольно проста в использовании:

```
// remove.cpp: Функция-член remove класса list.
#include <iostream>
#include <list>
using namespace std;

void out(const char *s, const list<int> &L)
{  cout << s;
   copy(L.begin(), L.end(),
      ostream_iterator<int>(cout, " "));
   cout << endl;
}

int main()
{  list<int> L;
   list<int>::iterator new_end;
   L.push_back(1); L.push_back(4); L.push_back(1);
   L.push_back(3); L.push_back(1); L.push_back(2);
   out("Initial sequence L:\n", L);
   L.remove(1);
   out("After L.remove(1):\n", L);
   return 0;
}
```

Вывод программы:

```
Initial sequence L:
1 4 1 3 1 2
After L.remove(1):
4 3 2
```

Функция-член *reverse* класса *list*

Класс *list* определяет также функцию *reverse*, объявленную следующим образом:

```
void reverse();
```

Эта функция-член, определенная только для списков, но не для векторов и двусторонних очередей, использует в своих целях устройство списка. Поэтому для списка *L* предпочтительнее употреблять вызов функции-члена

```
L.reverse();
```

нежели вызов алгоритма *reverse*

```
reverse(L.begin(), L.end());
```

Функция-член *merge* класса *list*

Объединение для списков может быть выполнено более эффективно, чем для векторов или двусторонних очередей, поскольку требуется копировать только ссылки, а не сами элементы. Класс *list* содержит следующее объявление функции-члена *merge*:

```
void merge(list<T>& x);
```

Приведенная далее программа показывает, как использовать эту функцию:

```
// lstmerge.cpp: Объединение списков
//               с помощью функции-члена merge.
#include <iostream>
#include <list>

using namespace std;

void out(const char *s, const list<int> &L)
{  cout << s;
   copy(L.begin(), L.end(),
ostream_iterator<int>(cout, " "));
cout << endl;
}

int main()
{  list<int> L1, L2, L3;
   list<int>::iterator new_end;
   L1.push_back(10); L1.push_back(20); L1.push_back(30);
   L2.push_back(15); L2.push_back(35);
   out("Initial sequence L1:\n", L1);
   out("Initial sequence L2:\n", L2);
   L1.merge(L2);
   out("After L1.merge(L2):\n", L1);
   return 0;
}
```

Эта программа выводит:

```
Initial sequence L1:
10 20 30
Initial sequence L2:
15 35
```

```
After L1.merge(L2):
10 15 20 30 35
```

Последняя строчка вывода показывает новый список $L1$, в то время как $L2$ становится пустым. Хотя имя функции *merge* совпадает с именем алгоритма *merge*, последний работает по-другому (см. раздел 1.7).

3.6. Векторы векторов

До сих пор в наших примерах излюбленным типом был *vector<int>*. Вполне очевидно, что мы можем заменить *int* на другой тип и использовать *vector<double>*, *vector<char>* и т. п. А вправе ли мы использовать более сложный тип, чем *int*, *double* и *char,* между двумя угловыми скобками? Например, можно ли написать

```
vector<vector<int> > A;
```

Да, возможно, как в следующей программе:

```
// vecvec.cpp: Вектор векторов.

#include <iostream>
#include <vector>

using namespace std;

int main()
{  vector <int> v;
   v.push_back(8); v.push_back(9);
   vector<vector<int> > A;
   A.push_back(v);
   A.push_back(v);
   A.push_back(v);
   for (int i=0; i<3; i++)
   {  for (int j=0; j<2; j++)
         cout << A[i][j] << " ";
      cout << endl;
   }
   return 0;
}
```

Программа использует вектор A, состоящий из трех элементов, каждый из которых является вектором, содержащим два целых числа 8 и 9. Мы можем считать вектор A двумерным массивом, или *матрицей*, как показывает вывод этой программы:

```
8 9
8 9
8 9
```

Однако такой подход может привести к тому, что нам потребуется больше памяти, чем действительно необходимо. Довольно часто программисты на C и C++ используют вместо двумерных массивов массивы указателей. Точно так же можно использовать вектор указателей, реализованный в следующей программе, вывод которой совпадает с предыдущей:

```
// pointers.cpp: Вектор указателей.
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> vecint;

int main()
{  vector<vecint*> A;
   int a[2] = {8, 9};
   A.push_back(new vecint(a, a+2));
   A.push_back(new vecint(a, a+2));
   A.push_back(new vecint(a, a+2));
   for (int i=0; i<3; i++)
   {  for (int j=0; j<2; j++)
         cout << (*A[i])[j] << " ";
      cout << endl;
   }
   delete A[0]; delete A[1]; delete A[2];
   return 0;
}
```

Каждый элемент вектора A, $A[i]$ является указателем на вектор из двух элементов целого типа, так что $*(A[i])$ является этим вектором, содержащим целые значения, доступ к которым осуществляется с помощью выражения $*(A[i])[j]$[1].

3.7. Как избавиться от явного выделения памяти

Как было отмечено в разделе 1.14, в сложных программах к частым ошибкам приводит освобождение памяти с помощью *free* или *delete*. В данных ниже определениях функций нельзя обойтись без использования упомянутых

[1] Следует иметь в виду, что предложенный автором второй вариант реализации двумерного массива с помощью вектора указателей не только менее эффективен по времени выполнения и менее защищен от случайного неправильного выделения/освобождения памяти (см. раздел 3.7), но в случае плотно заполненной матрицы может фактически использовать *больше* памяти, чем первый вариант, использующий вектор из векторов! В реальных программах на C++ подобных приемов следует избегать. – *Прим. переводчика.*

выражений. Если же мы забудем сделать это, возникнет так называемая *утечка памяти*, то есть динамически выделенная память не будет освобождена:

```
void f(int n)
{  double *a;
   a = (double*)malloc(n * sizeof(double));
   ...            // Используем a[0], ..., a[n-1].
   free(a);       // Во избежание утечки памяти
}
```

либо

```
void f(int n)
{  double *a;
   a = new double[n];
   ...            // Используем a[0], ..., a[n-1].
   delete[] a;    // Во избежание утечки памяти
}
```

Нам следует также заботиться о том, чтобы не разместить область памяти более одного раза и, если память была выделена при выполнении какого-либо условия, не пытаться освободить ее без проверки условия.

Используя STL, мы можем избежать использования *malloc* и *new*, применяя контейнер *vector*. Преимущество этого подхода заключается в снижении риска неправильного использования *free* или *delete*. Например, вместо рассмотренной выше функции мы можем написать

```
#include <vector>
...
void f(int n)
{  vector<double> a(n);      // См. раздел 3.3.
...                                 // Используем a[0], ...,
a[n-1].
}
```

Освобождение памяти теперь осуществляется деструктором контейнера.

Векторы как члены классов

Векторы также можно использовать вместо определенных внутри классов массивов, которые размещаются и освобождаются явным образом. Рассмотрим для примера следующий код, использующий явное выделение и освобождение памяти:

```
class T {
public:
   T(int n){a = new double[n];}
```

```
    ~T(){delete[] a;}
    ...
    double *a;
};
```

Здесь мы опять пишем *delete*[] *a* для предотвращения утечки памяти. Если для выделения памяти использовать конструктор вектора, то можно обойтись без явного освобождения памяти, как демонстрирует следующая программа:

```
// vecmem.cpp: Вектор - член класса.
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

template <class eltype>
class objtype {
public:
   objtype(int n=0): a(n){}
   vector<eltype> a;
};

int main()
{  int n, i;
   double s=0;
   cout << "Enter n: ";
   cin >> n; // n is a positive integer
   objtype<double> x(n);
   for (i=0; i<n; i++) x.a[i] = i;
   s = accumulate(x.a.begin(), x.a.end(), s);
   cout  << s << " = " << double(n-1)*n/2
         << endl;
   return 0;
}
```

Объект *x* содержит массив *a* из *n* значений типа *double*, где *n* вводится пользователем. Для заполнения этого массива мы используем значения 0, 1, ..., $n - 1$. Далее вычисляем сумму элементов массива, применяя алгоритм *accumulate*, который был рассмотрен в разделе 2.1. Другими словами, мы получим

$$0 + 1 + 2 + \ldots + n - 1 = {}^{1}/_{2}\, n\,(n - 1)$$

Указанная сумма вычисляется двумя способами, так что легко проверить результат, полученный с помощью алгоритма *accumulate*.

Вот пример использования этой программы для нахождения значения $^1/_2 \times 100000 \times 99999 = 4999950000 = 4.99995 \times 10^9$:

```
Enter n: 100000
4.99995e+09 = 4.99995e+09
```

Не требуется определять конструкторы копирования и операторы присваивания

Еще одним преимуществом использования вектора в качестве члена класса является то, что копирование и присваивание экземпляров класса будет осуществляться правильно без необходимости определения конструктора копирования и оператора присваивания. Следующая программа, основу которой составляет только что использованный класс, иллюстрирует это поведение:

```
// simple.cpp: Не нужен ни конструктор копирования,
//                  ни оператор присваивания.
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

template <class eltype>
class objtype {
public:
   objtype(int n=0): a(n, 0){ }
   vector<eltype> a;
};

int main()
{  int i;
   double s;
   objtype<double> x(3), y(x);
   // Или, если использовать присваивание,
   // а не инициализацию:
   //    objtype<double> x(3), y;
   //    y = x;
   x.a[1] = 123;

   copy(x.a.begin(), x.a.end(),
      ostream_iterator<double>(cout, " "));
   cout << endl;
   copy(y.a.begin(), y.a.end(),
      ostream_iterator<double>(cout, " "));
   cout << endl;
   return 0;
}
```

Вывод

```
0 123 0
0 0 0
```

этой программы со всей очевидностью показывает, что объекты *x* и *y* не зависят друг от друга: начальное значение *y* (0, 0, 0) никуда не исчезает, когда присваиваем *x* другое значение. Если бы мы выделили память в конструкторе *objtype* с помощью оператора *new*, проведение инициализации, как выше, с использованием «глубокого» копирования начального значения *x* для *y* потребовало бы написать конструктор копирования. Для приведенной версии программы в этом нет необходимости, поскольку используется конструктор копирования класса *vector*, определенный в STL. Точно так же нет необходимости писать свой оператор присваивания, что отмечено в комментарии к тексту программы.

4

Ассоциативные контейнеры

4.1. Введение

Из раздела 2.5 мы узнали, что существуют четыре типа ассоциативных контейнеров: множества, множества с дубликатами, словари и словари с дубликатами. Множества и множества с дубликатами характеризуются двумя параметрами шаблона, а словари и словари с дубликатами – тремя:

```
template <class Key, class Compare>
class set { ...
};

template <class Key, class Compare>
class multiset { ...
};

template <class Key, class T, class Compare>
class map { ...
};

template <class Key, class T, class Compare>
class multimap { ...
};
```

Полностью определения этих шаблонных классов помещены в заголовках *set* и *map*.

О переносимости

Если вы используете старую версию HP STL, вам придется включать заголовки *set.h* и *map.h* для множеств и словарей, а для вариантов этих контейнеров с дубликатами – соответственно *multiset.h* и *multimap.h*.

Сравнение и прочие функциональные объекты

Программисты, начинающие изучать STL, могут посчитать функциональные объекты трудными для понимания. Это отчасти вызвано применением сложных конструкций языка C++, которые редко используются большинством программистов. В дополнение к нашему обсуждению в разделе 1.12 рассмотрим эту тему более подробно. Начнем с напоминания, что выражение

```
int()
```

имеет в качестве значения целочисленную константу 0, отсюда следует, что два нижеприведенных выражения эквивалентны:

```
x = 0;
x = int();
```

Для нестандартного типа, например *T*, выражение *T*() возвращает объект типа *T*, как показывает следующая программа:

```
// gen_t.cpp: Создаем объект T().
#include <iostream.h>

class T {
public:
   T(){i = 123;}
   int i;
};

int main()
{  int j = T().i;
   cout << j << endl; // Вывод: 123
   return 0;
}
```

Вспомним, что, если за именем типа (в нашем примере это *T* в функции *main*) следует пара круглых скобок без параметров, это приводит к вызову *конструктора по умолчанию* для создания объекта. Конструктор по умолчанию не имеет параметров, либо все его параметры имеют значение по умолчанию. В этом примере мы определяем такой конструктор для класса *T*. Если бы мы этого не сделали, а также не определили бы никаких других конструкторов для *T*, компилятор создал бы за нас конструктор по умолчанию, эквивалентный *T*(){}.

По-другому обстоят дела, когда круглые скобки следуют не за именем типа, а после объекта; например, когда для объекта *u* типа *U* мы используем выражения вроде следующих:

```
u()
u(1, 2, 3)
```

Оба этих выражения разрешены только в том случае, если мы определяем соответствующие операторы вызова функции, описываемые как *operator*(), например:

```
// call_op.cpp: Два оператора вызова.

#include <iostream.h>

class U {
public:
   char operator()()
   {  return 'Q';
   }
   int operator()(int a, int b, int c)
   {  return a + b + c;
   }
};

int main()
{  U u;
   cout << "u() = " << u() << endl;
   cout << "u(1, 2, 3) = " << u(1, 2, 3) << endl;
   return 0;
}
```

Результат этой программы следующий:

```
u() = Q
u(1, 2, 3) = 6
```

В программе *call_op.cpp* определены два оператора вызова. Первый не имеет параметров и возвращает тип *char*, а второй принимает три параметра и возвращает тип *int*.

Схожесть выражений *T*() и *u*() в двух наших последних программах обманчива. Поскольку *T* – это тип, а *u* – объект, выражение *T*() стоит ближе по смыслу к выражению *u*, когда *T* сопровождается скобками, а *u* нет; как *T*(), так и *u* являются объектами. Если бы мы определили для класса *T* те же операторы вызова, что и для *U*, тогда выражения

```
T()()
T()(1, 2, 3)
```

имели бы смысл.

В следующей программе *LessThan* является типом, как *T* в предыдущем обсуждении, и поэтому *LessThan*() представляет собой объект, а *LessThan*()(2, 3) обозначает вызов функции:

```
// lessthan.cpp: Функциональный объект LessThan.
#include <iostream.h>

class LessThan {
public:
   int operator()(int x, int y)
   {  return x < y;
   }
};

int main()
{  LessThan b;
   cout  << "b(2, 3) = " << b(2, 3) << endl;
   cout  << "LessThan()(2, 3) = " << LessThan()(2, 3)
         << endl;
   return 0;
}
```

Эта программа печатает

```
b(2, 3) = 1
LessThan()(2, 3) = 1
```

Мы использовали класс *LessThan* для сравнения целых чисел. Можно сделать этот класс более общим, определив шаблонный класс *less_than* для сравнения двух объектов, принадлежащих к одному типу, для которого определен оператор <, как в следующей программе:

```
// lt_templ.cpp: Шаблонный класс для сравнения по условию "меньше".
#include <iostream.h>

template <class T>
class less_than {
public:
   int operator()(const T &x, const T &y)
   {  return x < y;
   }
};

int main()
{  less_than<int> b;
   cout << "b(2, 3) = " << b(2, 3) << endl;
   cout  << "less_than<double>()(2.1, 2.2) = "
         << less_than<double>()(2.1, 2.2) << endl;
   return 0;
}
```

Результат программы:

```
b(2, 3) = 1
less_than<double>()(2.1, 2.2) = 1
```

В последней строчке *less_than<double>* является типом, а *less_than<double>*() – объектом этого типа. Наконец, *less_than<double>*() (2.1, 2.2) является вызовом функции-члена *operator*() этого объекта.

Все это облегчит понимание выражений наподобие

```
settype S1(less<int>());
```

которые встретятся нам в следующем разделе. Приведенный пример является определением переменной *S*1. В этом определении объект *less<int>*() (типа *less<int>*) передается в качестве параметра конструктору класса *settype*.

4.2. Функции-члены множеств

У класса *set* имеются три конструктора, которые могут быть определены следующим образом внутри класса:

```
set(const Compare& comp = Compare()); // 1 (по умолчанию)
set(const value_type* first, const value_type* last,
   const Compare& comp = Compare());  // 2
set(const set<Key, Compare>& x);      // 3 (копирования)
```

Идентификатор *Compare* в приведенных объявлениях обозначает параметр шаблона, который является типом. Как мы показали в предыдущем разделе, это означает, что *Compare*() является объектом. Следующая программа использует все три конструктора:

```
// setconst.cpp: Конструкторы для множеств.

#include <iostream>
#include <set>

using namespace std;

typedef set<int, less<int> > settype;

void out(const char *s, const settype &S)
{  cout << s;
   copy(S.begin(), S.end(),
      ostream_iterator<int>(cout, " "));
   cout << endl;
}

int main()
{  int a[3] = {20, 10, 20};
```

```
    settype S1;                          // 1
    settype S2(a, a+3);                  // 2
    settype S3(S2);                      // 3
    out("S1: ", S1);
    out("S2: ", S2);
    out("S3: ", S3);
    return 0;
}
```

Несмотря на то что для конструирования *S*2 используется диапазон [*a*, *a* + 3) из трех элементов, в множество попадают только значения *a*[0] и *a*[1], из-за того что значение третьего элемента массива, *a*[2], равно *a*[0]. Вывод программы:

```
S1:
S2: 10 20
S3: 10 20
```

Аргумент, задающий функцию сравнения, нами опущен, поэтому используется значение по умолчанию *less<int>*(). Другими словами, программа имела бы то же самое действие, если бы мы написали:

```
settype S1(less<int>());                 // 1
settype S2(a, a+3, less<int>());         // 2
```

Обратите внимание, что третий конструктор (помеченный комментарием `// 3`) является конструктором копирования, который используется не только для инициализации переменных, но и для копирования аргументов (если значения передаются не по ссылке), и в операторе *return*.

Следующие две функции-члена возвращают функциональный объект для сравнения элементов множества. Оба типа *key_compare* и *value_compare* являются синонимами параметра *Compare* шаблонного класса *set*.

```
key_compare key_comp() const;
value_compare value_comp() const;
```

Приведенный ниже пример, хотя и не является слишком реалистичным, показывает применение этих функций:

```
cout  << "settype::value_compare()(2, 3) = "
      << settype::value_compare()(2, 3) << endl;
```

Если бы мы вставили такое непривычное выражение в функцию *main* программы *setconst.cpp*, то получили бы следующую строчку на выводе:

```
settype::value_compare()(2, 3) = 1
```

Перечисленные ниже функции-члены аналогичны по своему назначению одноименным функциям последовательных контейнеров:

```
iterator begin() const;
iterator end() const;
iterator rbegin() const;
iterator rend() const;
bool empty() const;
size_type size() const;
size_type max_size() const;
void swap(set<Key, Compare>& x);
```

В программе *setconst.cpp* *S*1.*size*() = 0, а *S*2.*size*() = 2. Как обычно, функция *max_size*() возвращает очень большое значение, например 3 214 748 364.

В разделе 2.10 на примере словарей мы рассматривали функцию *insert*, которая возвращает пару, составленную из итератора и булевского значения. Такая же функция существует и для множеств, и объявлена она так:

```
pair<iterator, bool> insert(const value_type& x);
```

Возвращаемое функцией значение содержит итератор, указывающий на позицию значения *x* после добавления в рассматриваемое множество, а также значение типа *bool*, которое равно *true* в том случае, если произошло добавление *x* ко множеству, или *false*, если значение *x* уже присутствовало в множестве.

Вторая функция *insert* в качестве первого аргумента принимает итератор; этот итератор используется как подсказка, указывая на позицию, после которой следует добавить новое значение. Вставка происходит быстрее всего, если имеется возможность добавить новое значение непосредственно после позиции, на которую указывает первый аргумент. Эта функция объявлена следующим образом:

```
iterator insert(iterator position, const value_type& x);
```

И наконец, существует функция *insert*, которая позволяет добавить сразу диапазон значений:

```
void insert(const value_type* first,
   const value_type* last);
```

Мы можем использовать эту функцию, когда нужно добавить в множество последовательный ряд элементов массива, как, например, в следующей программе:

```
// insrange.cpp: Добавление диапазона значений.
#include <iostream>
#include <set>
```

```
using namespace std;

int main()
{  int a[3] = {20, 10, 20};
   set<int, less<int> > S;
   S.insert(a, a+2);
   copy(S.begin(), S.end(),
   ostream_iterator<int>(cout, " "));
   cout << endl;      // Вывод 10 20
   return 0;
}
```

Для удаления элементов множества определены три функции:

```
void erase(iterator position);              // 1
size_type erase(const key_type& x);         // 2
void erase(iterator first, iterator last);  // 3
```

Если мы знаем позицию удаляемого элемента, рекомендуется использовать первую функцию. Если известно только значение этого элемента, мы используем вторую; эта функция возвращает количество удаленных элементов, как правило 1, но может равняться 0, если нужное значение отсутствует в множестве. Последняя из трех функций удаляет диапазон значений [*first*, *last*). Можно, следовательно, удалить все элементы множества *S* и сделать *S* пустым, написав

```
S.erase(S.begin(), S.end());
```

Поскольку элементы множества уникальны (не может быть двух равных элементов), функция *count*, объявленная как

```
size_type count(const key_type& x) const;
```

возвратит либо 0, либо 1. Например, если множеством целых чисел является *S*, выражение *S.count*(123) принимает значение 1, когда 123 входит в множество *S*, и 0, когда не входит.

Следующие три функции осуществляют поиск заданного значения *x* в множестве:

```
iterator find(const key_type& x) const;
iterator lower_bound(const key_type& x) const;
iterator upper_bound(const key_type& x) const;
```

Как обычно, после выполнения

```
i = S.find(x);
```

значение i равно *S.end*(), если x не найдено, в противном случае $*i$ равняется x. Чтобы найти минимальный диапазон $[i, j)$ из множества S, содержащий x, мы можем написать

```
i = S.lower_bound(x);
j = S.upper_bound(x);
```

Для примера предположим, что S состоит из целых чисел 30, 40 и 50. Таблица ниже содержит значения i и j для разных значений x. Строки, в которых x = 25, 35, 45 и 55, показывают, что вызовы функций *S.lower_bound*(x) и *S.upper_bound*(x) возвращают равные значения итераторов, если S не содержит x. Напротив, значения 30, 40 и 50 показывают, что возвращаемые итераторы различны, когда S содержит x:

x	*i = S.lower_bound(x);*	*j = S.upper_bound(x);*
25	$*i = 30$	$*j = 30$
30	$*i = 30$	$*j = 40$
35	$*i = 40$	$*j = 40$
40	$*i = 40$	$*j = 50$
45	$*i = 50$	$*j = 50$
50	$*i = 50$	*j = S.end*()
55	*i = S.end()*	*j = S.end()*

Функция *equal_range* возвращает рассматриваемую пару (i, j) за один вызов. Объявление этой функции дано ниже:

```
pair<iterator, iterator> equal_range(const key_type& x)
   const;
```

Вместо того чтобы вызывать отдельно *lower_bound* и *upper_bound*, как мы делали это выше с помощью операторов

```
i = S.lower_bound(x);
j = S.upper_bound(x);
```

более эффективно находить i и j с помощью *equal_range*, как в приведенной ниже программе:

```
// eqrange.cpp: Функция-член equal_range.
#include <set>
#include <iostream>
using namespace std;

int main()
{  typedef set<int, less<int> > settype;
   typedef settype::iterator iterator;
   settype S;
```

```
    S.insert(30);
    S.insert(40);
    S.insert(50);
    pair<iterator, iterator> P;
    iterator i, j;
    int x = 30;
    P = S.equal_range(x);
    i = P.first; j = P.second;
    cout << *i << " " << *j << endl; // Вывод: 30 40
    return 0;
}
```

4.3. Объединение и пересечение множеств

В этом разделе мы рассмотрим известные математические операции нахождения пересечения и объединения двух множеств, которые проиллюстрированы на рисунке 4.1.

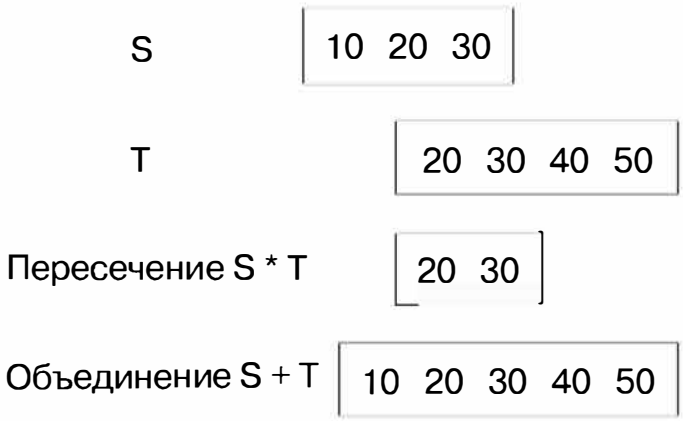

Рисунок 4.1. Пересечение и объединение двух множеств

Для обозначения операций пересечения и объединения мы будем использовать операторы * и +, хотя в математике обычно используются обозначения $\cap$ и $\cup$. Для двух множеств S и T пересечение S * T определяется как множество, состоящее из элементов, входящих одновременно в оба множества S и T, а объединение $S + T$ – как множество, содержащее все элементы, входящие либо в S, либо в T, либо в оба этих множества. Следующая программа показывает, как можно определить операторы * и + для множеств:

```
// intunio1.cpp: Пересечение и объединение множеств.
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
typedef set<int, less<int> > settype;

settype operator*(const settype &S, const settype &T)
```

```
{  settype I;  // Пересечение.
   set_intersection( S.begin(), S.end(),
                     T.begin(), T.end(),
                     inserter(I, I.begin()));
   return I;
}

settype operator+(const settype &S, const settype &T)
{  settype U;  // Объединение
   set_union(  S.begin(), S.end(),
               T.begin(), T.end(),
               inserter(U, U.begin()));
   return U;
}

void out(const char *s, const settype &S)
{  cout << s;
   copy(S.begin(), S.end(),
   ostream_iterator<int>(cout, " "));
   cout << endl;
}

int main()
{  int aS[3] = {10, 20, 30}, aT[4] = {20, 30, 40, 50};
   settype  S(aS, aS + 3),
            T(aT, aT + 4);
   out("S =     ", S);
   out("T =     ", T);
   out("S * T = ", S * T);
   out("S + T = ", S + T);
   return 0;
}
```

В соответствии с примером на рисунке 4.1 программа выводит результаты:

```
S =     10 20 30
T =     20 30 40 50
S * T = 20 30
S + T = 10 20 30 40 50
```

Определение операторов * и + с помощью функции *insert* вместо алгоритмов *set_intersection* и *set_union* является полезным упражнением. Такие варианты этих операторов приведены ниже; если их использовать в рассмотренной выше программе, результат ее работы не изменится:

```
// Две функции, которые мы могли бы использовать,
// если бы не существовало алгоритмов set_union
// и set_intersection:

settype operator*(const settype &S, const settype &T)
```

```
{  settype I;  // Пересечение
   settype::iterator
      firstS = S.begin(), lastS = S.end(),
      firstT = T.begin(), lastT = T.end(),
      i = I.begin();
   while (firstS != lastS && firstT != lastT)
   {  if (*firstS < *firstT) ++firstS; else
      if (*firstT < *firstS) ++firstT; else
      {  i = I.insert(i, *firstS++);
         ++firstT;
      }
   }
   return I;
}

settype operator+(const settype &S, const settype &T)
{  settype U;  // Объединение
   settype::iterator
      firstS = S.begin(), lastS = S.end(),
      firstT = T.begin(), lastT = T.end(),
      i = U.begin();
   while (firstS != lastS && firstT != lastT)
      i = U.insert(i,
         (*firstS < *firstT ? *firstS++ : *firstT++));
   while (firstS != lastS) i = U.insert(i, *firstS++);
   while (firstT != lastT) i = U.insert(i, *firstT++);
   return U;
}
```

Безусловно, предпочтительнее использовать оригинальные, более простые версии этих функций.

Для нахождения пересечения и объединения двух множеств нет необходимости, разумеется, применять операторы * и +, что показывает следующая программа, эквивалентная рассмотренной выше; данная версия более эффективна, так как не требует копирования возвращаемых функциями множеств:

```
// intunio2.cpp: Пересечение и объединение множеств.
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
typedef set<int, less<int> > settype;

void out(const char *s, const settype &S)
{  cout << s;
   copy(S.begin(), S.end(),
```

```
    ostream_iterator<int>(cout, " "));
    cout << endl;
}

int main()
{  int aS[3] = {10, 20, 30}, aT[4] = {20, 30, 40, 50};
   settype  S(aS, aS + 3),
            T(aT, aT + 4), prod, sum;
   out("S =     ", S); out("T =     ", T);
   set_intersection(S.begin(), S.end(),
      T.begin(), T.end(), inserter(prod, prod.begin()));
   out("S * T = ", prod);
   set_union(S.begin(), S.end(),
      T.begin(), T.end(),inserter(sum, sum.begin()));
   out("S + T = ", sum);
   return 0;
}
```

Алгоритмы *set_intersection* и *set_union* не ограничены в использовании контейнерами типа множеств, а могут быть применены и к другим сортированным структурам, как показано далее в разделе 7.3.8.

4.4. Отличия множеств с дубликатами от просто множеств

Мы уже обсуждали множества и множества с дубликатами в разделе 2.6, и не мешает снова обратиться к тексту программы *multiset.cpp,* приведенному в том разделе. Напомним, что каждый элемент множества уникален, в то время как множества с дубликатами могут содержать несколько экземпляров одинаковых элементов. Из-за этого одна из функций *insert* для множеств с дубликатами определена проще, чем для множеств. Как видно из раздела 4.2, для множеств существует функция *insert*, объявленная как

```
pair<iterator, bool> insert(const value_type& x); // set
```

Вспомним, что эта функция возвращает пару, состоящую из итератора и значения типа *bool*, которое равно *false*, когда операция добавления завершается неудачей из-за того, что значение *x* уже присутствует в множестве. Напротив, добавление элемента в множество с дубликатами всегда проходит успешно, так что нет необходимости возвращать признак типа *bool*. Поэтому соответствующая функция для множества с дубликатами возвращает просто итератор в соответствии со следующим объявлением:

```
iterator insert(const value_type& x); // multiset
```

Все остальные функции для множеств с дубликатами объявлены точно так же, как и функции для множеств, если не считать отличием замену имени класса *set* в объявлении на *multiset*. Функция *find* для множеств

с дубликатами в случае успеха возвращает итератор, ссылающийся на *первый* из элементов с искомым значением, как видно из программы:

```
// msfind.cpp: Поиск в множестве с дубликатами.
#include <iostream>
#include <set>

using namespace std;

typedef multiset<int, less<int> > multisettype;
typedef multisettype::iterator Iterator;

void out(Iterator first, Iterator last)
{  for (Iterator i = first; i != last; ++i)
      cout << *i << " ";
   cout << endl;
}

int main()
{  int a[5] = {10, 20, 20, 20, 30};
   multisettype M(a, a + 5);
   out(M.begin(), M.end());
   cout << "Subrange starting at element 20:\n";
   out(M.find(20), M.end());
   return 0;
}
```

Программа выводит

```
10 20 20 20 30
Subrange starting at element 20:
20 20 20 30
```

Из этого видно, что возвращаемый функцией *find* итератор указывает на первый найденный элемент. Если бы вызов *M.find*(20) завершился неудачей из-за отсутствия в множестве с дубликатами *M* элемента со значением 20, этот вызов возвратил бы значение *M.end*().

4.5. Словари

У словарей существуют три конструктора, объявленные следующим образом:

```
map(const Compare& comp = Compare());                    // 1
map(const value_type* first, const value_type* last,
    const Compare& comp = Compare());                    // 2
map(const map<Key, T, Compare>& x);                      // 3
```

Программа ниже показывает, как используется каждый из этих конструкторов:

```
// mapcstr.cpp: Конструкторы словаря
#include <iostream>
#include <map>
using namespace std;
```

```
typedef map<int, double, less<int> > maptype;
typedef pair<int, double> Pair;

int main()
{  pair<int, double> a[3] =
   {  Pair(20, 1.5),
      Pair(800, 0.3),
      Pair(3, 0.2)
   };
   maptype MA;                // 1 (начинаем с пустого словаря)
   maptype MB(a, a + 3);      // 2 (инициализируем массивом)
   maptype MC(MB);            // 3 (используем MB для
                              //    инициализации)
   cout << MC[800] << endl;   // MC[800] = MB[800] = 0.3
   return 0;
}
```

Инициализация словаря *MB* записывается проще, чем инициализация массива пар *a*, который служит для нее основой. С помощью конструктора копирования (помеченного комментарием // 3 в начале этого раздела) мы определяем словарь *MC* как копию словаря *MB*.

Для словаря очень полезен оператор доступа по индексу []. Мы можем использовать его не только для извлечения данных из словаря, как в предыдущей программе для словаря *MC*, но и для добавления новых данных.

Например, вместо использования массива *a* в предыдущей программе мы могли бы добавить данные в *MB* более простым способом, как в следующей программе:

```
// mapsubs.cpp: Использование индекса для словаря.
#include <iostream>
#include <map>
using namespace std;

typedef map<int, double, less<int> > maptype;

int main()
{  maptype MB;
   MB[20] = 1.5;
   MB[800] = 0.3;
   MB[3] = 0.2;
   cout << MB[800] << endl; // 0.3
   return 0;
}
```

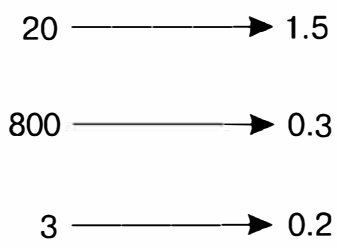

Рисунок 4.2. Отображение ключей на значения

Доступ по индексу является отображением ключей (20, 800 и 3) на связанные с ними значения (1.5, 0.3 и 0.2), как показано на рисунке 4.2.

При доступе по индексу к обычному массиву мы используем последовательные значения индекса 0, 1,

2 и т. д., которые нет необходимости сохранять где-либо, поскольку элементы массива размещаются непрерывно друг за другом. В противоположность этому как ключи, так и соответствующие значения словарей хранятся в сбалансированном двоичном дереве поиска (см. пример на рис. 4.3).

В двоичных деревьях поиска удобно искать ключи (на рисунке 4.3 они подчеркнуты) из-за принципа построения таких деревьев: для каждого узла все ключи в левом поддереве меньше ключа в этом узле, а в правом поддереве все ключи больше.

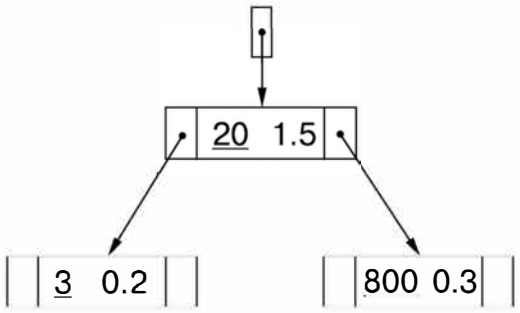

Рисунок 4.3. Сбалансированное двоичное дерево поиска, служащее для представления словаря

Функции-члены *insert* для словарей

Вместо доступа по индексу мы можем добавлять элементы словарей с помощью трех функций-членов *insert*, объявленных как:

```
pair<iterator, bool> insert(const value_type& x);
interator insert(iterator position, const value_type& x);
void insert(const value_type* first,
   const value_type* last);
```

Они похожи на функции *insert* для множеств, о которых говорилось в разделе 4.2, но в этом объявлении тип *value_type* более сложный, потому что каждый элемент является парой (ключ, значение). Как видно из программы *mapcstr.cpp* в предыдущем разделе, мы можем использовать конструктор класса *pair<int, double>* для создания такой пары. Например, вместо

```
MB[800] = 0.3;
```

можно написать

```
MB.insert(pair<int, double>(800, 0.3));
```

не используя значение, возвращаемое функцией *insert*. Однако добавления с помощью этой функции не происходит, если ключ уже присутствует в словаре.

Следующая программа показывает, как можно использовать возвращаемое этой функцией значение. Она создает словарь, содержащий только один элемент (800, 0.3). Попытка заменить этот элемент на элемент (800, 0.7) не приводит к успеху, потому что их ключи совпадают:

```
// mapins.cpp: Значение, возвращаемое функцией insert
//             класса map.
#include <iostream>
#include <map>
using namespace std;

typedef map<int, double, less<int> > maptype;
typedef maptype::iterator Iterator;

void MyInsert(maptype &M, int k, double x)
{  pair<Iterator, bool>
      P = M.insert(make_pair(k, x));
   Iterator i = P.first;
   bool b = P.second;

   cout  << "After attempt to insert (" << k << ", " << x
         << "), returning P = (i, b):\n";
   cout  << "(*i).first = " << (*i).first << endl;
   cout  << "(*i).second = " << (*i).second << endl;
   cout  << "b = " << b << endl << endl;
}

int main()
{  maptype M;
   MyInsert(M, 800, 0.3);
   MyInsert(M, 800, 0.7);
   return 0;
}
```

Вывод этой программы выглядит следующим образом:

```
After attempt to insert (800, 0.3), returning P = (i, b):
(*i).first = 800
(*i).second = 0.3
b = 1

After attempt to insert (800, 0.7), returning P = (i, b):
(*i).first = 800
(*i).second = 0.3
b = 0
```

Каждая операция вставки в этой программе возвращает пару (*i*, *b*), где *i* – итератор, ссылающийся на вставленный элемент, а *b* – значение типа *bool*, показывающее, успешно ли прошла операция. Две последние строчки приведенного выше вывода программы показывают, что элемент (800, 0.7)

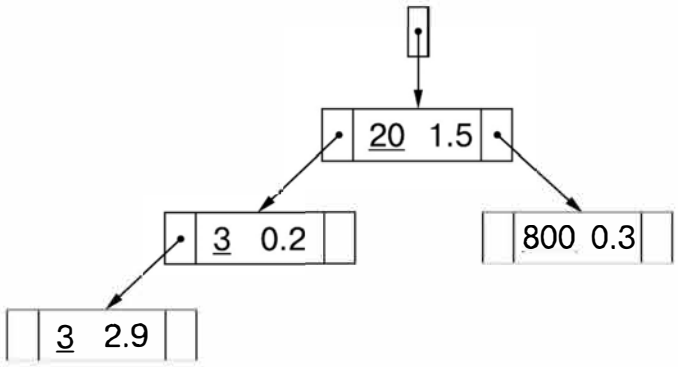

Рисунок 4.4. Сбалансированное двоичное дерево поиска, служащее для представления словаря с дубликатами

```
D["Johnson, J."] = 12345; // ???
```

Функции *insert* и *erase* для словарей с дубликатами объявлены следующим образом:

```
iterator insert(const value_type& x);
iterator insert(iterator position, const value_type& x);
void  insert(const value_type* first,
             const value_type* last);

void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);
```

Эти функции подобны своим аналогам для словарей, за исключением первой функции *insert*, которая возвращает просто итератор, ссылающийся на новый элемент. Так как добавление происходит успешно, даже если уже присутствует элемент с тем же ключом, нет необходимости возвращать дополнительно булевское значение, свидетельствующее об успехе операции.

Обратим внимание на порядок, в котором элементы с одинаковыми ключами добавляются в словарь с дубликатами. Его демонстрирует следующая программа:

```
// mmapins.cpp: Добавление элементов в словарь
//              с дубликатами.

#include <iostream>
#include <map>

using namespace std;

typedef multimap<int, double, less<int> > multimaptype;
typedef multimaptype::iterator Iterator;

void MyInsert(multimaptype &M, int k, double x)
{  Iterator i = M.insert(make_pair(k, x));
```

```
    cout << (*i).first << " " << (*i).second;
    // Эквивалентно:
    // cout << k << " " << x;
    cout << " inserted\n";
}

int main()
{  multimaptype M;
   MyInsert(M, 800, 0.3);
   MyInsert(M, 800, 0.7);
   MyInsert(M, 800, 0.5);
   MyInsert(M, 100, 1.9);
   MyInsert(M, 800, 0.6);
   cout << "Multimap traversal:\n";
   for (Iterator i = M.begin(); i != M.end(); ++i)
      cout << (*i).first << " " << (*i).second << endl;
   return 0;
}
```

Результат работы программы *mmapins.cpp* показывает, что порядок элементов с одинаковыми ключами в словаре с дубликатами совпадает с порядком, в котором они были добавлены в указанный контейнер. В противоположность этому разные ключи следуют в возрастающем порядке, вне зависимости от порядка их добавления.

```
800, 0.3 inserted
800, 0.7 inserted
800, 0.5 inserted
100, 1.9 inserted
800, 0.6 inserted
Multimap traversal:
100, 1.9
800, 0.3
800, 0.7
800, 0.5
800, 0.6
```

Вторая из трех функций *erase* (с одним аргументом, ключом) удаляет все элементы с заданным ключом и возвращает число удаленных элементов. Например, если мы в программе *mmapins.cpp* добавим строчки

```
int n = M.erase(800);
cout << n << " elements erased\n";
```

непосредственно после пяти вызовов функции *MyInsert*, вывод программы будет следующий:

```
800, 0.3 inserted
800, 0.7 inserted
800, 0.5 inserted
```

```
main            10
return          7 12
t               4 5 6
template        1 4
x               5 6 7
```

С помощью контейнера STL *map* (словарь) мы можем хранить все слова в сбалансированном двоичном дереве поиска (что делает поиск очень быстрой операцией), не программируя самостоятельно функциональность такого дерева. Каждый узел дерева содержит слово (используемое в качестве ключа) вместе с множеством номеров строк. В терминологии STL каждый элемент словаря является парой (*first*, *second*), где

first = слово; представляется типом ***string;***
second= множество номеров строк; представляется типом ***set<int>***.

Имя входного файла также будет содержаться в переменной типа *string*; полностью программа приведена ниже:

```
// concord.cpp: Сводный указатель, использующий словари,
//              множества и строки.
#include <iostream>
#include <fstream>
#include <iomanip>
#include <ctype.h>
#include <string>
#include <set>
#include <map>
using namespace std;

typedef set<int, less<int> > settype;
typedef map<string, settype, less<string> > maptype;

bool wordread(ifstream &ifstr, string &word,
   int &linenr)
{  char ch;
   // найдем первую букву:
   for (;;)
   {  ifstr.get(ch);
      if (ifstr.fail()) return false;
      if (isalpha(ch)) break;
      if (ch == '\n') linenr++;
   }
   word = "";
   // найдем первый небуквенный символ:
   do
   {  word += tolower(ch);
      ifstr.get(ch);
   }  while (!ifstr.fail() && isalpha(ch));
   if (ifstr.fail()) return false;
```

```
   ifstr.putback(ch); // ch может быть '\n'
   return true;
}

int main()
{  maptype M;
   maptype::iterator im;
   settype::iterator is, isbegin, isend;
   string inpfilename, word;
   ifstream ifstr;
   int linenr = 1;
   cout << "Enter name of input file: ";
   cin >> inpfilename;
   ifstr.open(inpfilename.c_str());
   if (!ifstr)
   {  cout << "Cannot open input file.\n"; exit(1);
   }

   while (wordread(ifstr, word, linenr))
   {  im = M.find(word);
      if (im == M.end())
         im = M.insert(maptype::value_type(word,
            settype())).first;
      (*im).second.insert(linenr);
   }
   for (im = M.begin(); im != M.end(); im++)
   {  cout  << setiosflags(ios::left) << setw(15)
            << (*im).first.c_str();
      isbegin = (*im).second.begin();
      isend = (*im).second.end();
      for (is=isbegin; is != isend; is++)
         cout << " " << *is;
      cout << endl;
   }
   return 0;
}
```

Функция *wordread* пропускает ненужные символы, увеличивает при необходимости счетчик строк, а затем читает одно слово. Хотя эта функция составляет значительную часть программы, мы не будем подробно ее обсуждать, потому что она не имеет непосредственного отношения к STL. Приведенный ниже фрагмент программы создает полный словарь, содержащий основные данные:

```
while (wordread(ifstr, word, linenr))
{  im = M.find(word);
   if (im == M.end())
      im = M.insert(maptype::value_type(word,
         settype())).first;
```

```
    (*im).second.insert(linenr);
}
```

Вызов *wordread*(*ifstr*, *word*, *linenr*) читает, если это возможно, следующее слово из потока *ifstr*. Он возвращает *true*, если слово было прочитано успешно, и *false*, если встретился конец файла. После этого мы с помощью функции *find* проверим, присутствует ли прочитанное слово в словаре. Теперь мы различаем два случая: слово найдено или не найдено. Вспомним, что элемент словаря состоит из слова и множества номеров строк. Если слово найдено в словаре, текущий номер строки добавляется в множество номеров строк; каким именно образом, мы увидим ниже. Если слово не найдено, возвращенный функцией итератор *im* равен *M.end*(), и сначала нам нужно добавить новый элемент словаря, состоящий из нового слова и пустого множества. Этого мы достигаем с помощью довольно сложного оператора, располагающегося на двух строках программы; он имеет следующий вид:

```
im = M.insert(xxx).first;
```

Рассматриваемый вызов возвращает пару (*iterator*, *true*), поскольку нам известно, что ключ пока отсутствует в словаре. Значение итератора из этой пары, указывающее на позицию только что добавленного элемента, присваивается переменной *im*. В приведенной выше записи *xxx* заменяет собой пару (*k*, *d*), где *k* является новым словом, а *d* – пустым множеством, которое записано в программе в виде выражения *settype*(). Напомним, что это выражение вызывает конструктор по умолчанию для типа *settype*, который является типом используемых нами множеств. Для пары (*k*, *d*) = *xxx* мы в действительности пишем

```
maptype::value_type(word, settype())
```

(тип *value_type* мы обсуждали в разделах 2.7, 2.10 и 3.1). Теперь в любом из двух упоминавшихся случаев *im* указывает на элемент словаря, который содержит рассматриваемое слово как (**im*).*first*. В соответствующее множество (**im*).*second* текущий номер строки добавляется с помощью оператора

```
(*im).second.insert(linenr);
```

5

Адаптеры контейнеров

5.1. Стеки

Стек (stack) представляет собой структуру данных, которая допускает только две операции, изменяющие ее размер: *push* (для добавления элемента в конце) и *pop* (для удаления элемента в конце). Иными словами, стек работает по принципу «последний пришел – первый ушел» (также называемому LIFO от английского Last In – First Out). Кроме *push* и *pop* для стека определены также функции-члены *empty* и *size*, имеющие обычное значение, и *top* (вместо *back*) для доступа к последнему элементу.

Стек может быть реализован с помощью каждого из трех последовательных контейнеров STL: вектора, двусторонней очереди и списка. Следовательно, стек не является новым типом контейнера, а особым вариантом вектора, двусторонней очереди либо списка, отсюда и происхождение термина *адаптер* контейнера.

В качестве примера используем стек для чтения последовательности целых чисел и отображения их в обратном порядке. Любой нецифровой символ будет признаком конца ввода. В следующей программе стек реализован вектором, но программа также будет работать, если мы заменим всюду *vector* на *deque* или *list*. Кроме того, программа показывает, как работают функции-члены *empty*, *top* и *size*.

```
// stack1.cpp: Используем стек для чтения
//             последовательности целых чисел произвольной длины
//             и отображения этой последовательности
//             в обратном порядке.

#include <iostream>
#include <vector>
#include <stack>
using namespace std;

int main()
{  stack <int, vector<int> > S;
   int x;
   cout <<
      "Enter some integers, followed by a letter:\n";
   while (cin >> x) S.push(x);
   while (!S.empty())
   {  x = S.top();
      cout << "Size: " << S.size()
         << "    Element at the top: " << x << endl;
      S.pop();
   }
   return 0;
}
```

Согласно проекту стандарта C++ шаблон *stack* имеет два параметра, как написано в приведенной программе:

```
stack <int, vector<int> > S;
```

В версии HP STL первый из этих аргументов отсутствует, поэтому, если вы используете эту (устаревшую) версию, необходимо данную строчку программы заменить на

```
stack < vector<int> > S;
```

После того как числа, введенные пользователем, будут положены на стек, программа последовательно нужное количество раз отобразит текущий размер стека и элемент, который будет удален следующим, как в приведенном примере:

```
Enter some integers, followed by a letter:
10 20 30 A
Size: 3    Element at the top: 30
Size: 2    Element at the top: 20
Size: 1    Element at the top: 10
```

Для стеков мы не можем использовать итераторы, а также не можем получить доступ к произвольному элементу стека без изменения его размера.

Интересно отметить, что функция-член *top* возвращает ссылку, которая позволяет нам изменить непустой стек, не прибегая к помощи операций *push* и *pop*. Например, вместо

```
S.pop();
S.push(15);
```

мы можем написать

```
S.top() = 15;
```

5.2. Очереди

Очередь (queue) является структурой данных, в которую можно добавлять элементы с одного конца, сзади, и удалять с другого конца, спереди. Мы можем узнать и изменить значения элементов спереди и сзади, как показано на рисунке 5.2.

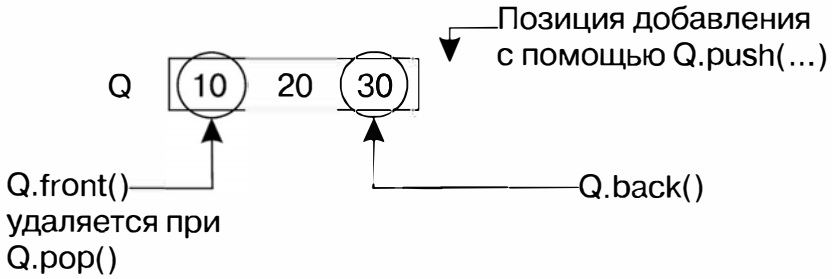

Рисунок 5.2. Очередь

В отличие от стека мы не можем представить очередь с помощью вектора, поскольку у вектора отсутствует операция *pop_front*. Например, нельзя написать

```
queue<int, vector<int> > Q;    // Ошибка
```

Но если *vector* заменить на *deque* или *list*, такая строчка станет допустимой. Из следующей программы видно, что функции-члены *push* и *pop* работают так, как показано на рисунке 5.2.

```
// queue.cpp: Использование очереди; демонстрация
//                  функций-членов push, pop, back и front.

#include <iostream>
#include <list>
#include <queue>

using namespace std;

int main()
{  queue <int, list<int> > Q;
```

```
    int x;
    P.push(123); P.push(51); P.push(1000); P.push(17);
    while (!P.empty())
    {  x = P.top();
       cout << "Retrieved element: " << x << endl;
       P.pop();
    }
    return 0;
}
```

В этой программе числа следуют в нисходящем порядке:

```
Retrieved element: 1000
Retrieved element: 123
Retrieved element: 51
Retrieved element: 17
```

Поскольку требуется проводить сравнение элементов, шаблон *priority_queue* имеет третий параметр, как видно из определения очереди с приоритетами *P*:

```
priority_queue <int, vector<int>, less<int> > P;
```

Если требуется извлекать элементы в порядке возрастания, мы можем просто заменить *less<int>* на *greater<int>*. При работе с HP STL необходимо опустить первый аргумент шаблона, а также использовать заголовок *stack.h* вместо *queue*.

Чтобы продемонстрировать возможность задания любого правила упорядочения элементов, приведем еще один пример, в котором элементы будут извлекаться по порядку возрастания последних цифр в десятичном представлении целых чисел, хранящихся в очереди с приоритетами:

```
// lastdig.cpp: Очередь с приоритетами; P.top() указывает
//              на элемент, последняя цифра которого не больше
//              последних цифр других элементов.

#include <iostream>
#include <vector>
#include <queue>
using namespace std;

class CompareLastDigits {
public:
   bool operator()(int x, int y)
   {  return x % 10 > y % 10;
   }
};
```

```
int main()
{  priority_queue <int, vector<int>, CompareLastDigits>
      P;
   int x;
   P.push(123); P.push(51); P.push(1000); P.push(17);
   while (!P.empty())
   {  x = P.top();
      cout << "Retrieved element: " << x << endl;
      P.pop();
   }
   return 0;
}
```

Вывод этой программы содержит добавленные целые числа 123, 51, 1000, 17 в порядке возрастания их последних цифр (0 < 1 < 3 < 7):

```
Retrieved element: 1000
Retrieved element: 51
Retrieved element: 123
Retrieved element: 17
```

Отметим, что нам необходимо использовать функциональный объект, поскольку шаблон *priority_queue* требует в качестве третьего параметра тип. Этот тип определяет идентификатор *CompareLastDigits*. Сравнение

```
x % 10 > y % 10
```

содержит оператор >, в результате чего элемент с наименьшей последней цифрой предшествует другим элементам. Аналогичным образом при использовании *greater<int>* первым будет располагаться наименьший элемент.

6

Функциональные объекты и адаптеры

6.2. Функциональные объекты

В разделах 1.12 и 2.4 уже рассмотрены функциональные объекты, которые зачастую бывают трудны для восприятия, поэтому мы немного поэкспериментируем с ними вне рамок STL. Следующая программа демонстрирует класс *sq*, который можно использовать для вычисления значения x^2 целого числа x:

```
// funobj1.cpp: Простой функциональный объект.
#include <iostream.h>

struct sq {
   int operator()(int x)const {return x * x;}
};

int main()
{  cout << "5 * 5 = " << sq()(5) << endl; // 25
   return 0;
}
```

Ключевое слово *struct* часто используется вместо *class,* когда класс имеет только открытые (public) члены. Если не считать доступ по умолчанию

(закрытый (private) для *class* и открытый для *struct*), эти два ключевых слова эквивалентны.

Из раздела 1.12 известно, что выражение *sq*() вызывает конструктор по умолчанию класса *sq*, так что значение этого выражения является экземпляром этого класса. Строка

```
int operator()(int x) {return x * x;}
```

в этом классе определяет *operator*() таким образом, что выражение *sq*()(5) является разрешенным вызовом функции, результат которого равен 25. Важно понимать, чем отличаются следующие выражения:

sq	класс (тип), который мы будем называть *функциональным классом;*
sq()	*функциональный объект;*
sq()(5)	вызов функции.

Функциональные объекты обладают всеми свойствами обычных функций, но они имеют дополнительные возможности, как показывает следующая программа:

```
// funobj2.cpp: Функциональные классы как
//              параметры шаблона.
#include <iostream.h>

struct square {
   int operator()(int x)const {return x * x;}
};

struct cube {
   int operator()(int x)const {return x * x * x;}
};

template <class T>
class cont {
public:
   cont(int i): j(i){}
   void print()const {cout << T()(j) << endl;}
private:
   int j;
};

int main()
{  cont<square> numsq(10);
   numsq.print();  // Вывод: 100
   cont<cube> numcub(10);
   numcub.print(); // Вывод: 1000
   return 0;
}
```

В функции *main* этой программы функциональные классы (*square* и *cube*) используются в качестве параметров шаблона. Мы можем считать, что *cont<T>* является ограниченным вариантом контейнера, поскольку он может хранить только одно число целого типа. Однако этот класс является довольно общим в том плане, что тип-параметр *T* может быть любым функциональным классом, функциональный вызов которого принимает целочисленный аргумент и возвращает целочисленное значение. В нашем примере такими классами являются *square* и *cube*. Хотя программа *funobj2.cpp* выглядит надуманной, она может помочь разобраться с более интересными случаями. Например, контейнер *priority_queue* также требует функциональный класс в качестве параметра, как мы видели в конце предыдущей главы.

Функции *operator*() имеют в нашем примере только один аргумент, хотя часто используются функции с двумя аргументами, как, например, в случае очередей с приоритетами. Приведем пример работы функциональных объектов с двумя аргументами функции. Определим шаблонный класс *PairSelect*, содержащий функцию печати, которая выводит меньший элемент пары в соответствии с определенным нами отношением «меньше чем», задаваемым параметром шаблона. Также в виде параметра шаблона мы зададим тип элементов пары. Следующая программа использует два отношения упорядочения. Они реализованы в виде *бинарных предикатов*, то есть функций, которые имеют два аргумента и возвращают логическое значение. Наш первый бинарный предикат, *LessThan,* является шаблоном, поэтому он применим к любому типу, для которого определен оператор <. Второй предикат, *CompareLastDigits*, является обычным, не шаблонным, функциональным классом, который практически совпадает с функциональным классом, определенным в последнем разделе предыдущей главы, только результат его вызова равен *true*, если последняя цифра первого аргумента меньше, чем у второго, и *false* – в противном случае.

```
// funobj3.cpp: Функция operator() как
//              бинарный предикат.
#include <iostream.h>

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

struct CompareLastDigits {
   bool operator()(int x, int y)const
   {  return x % 10 < y % 10;
   }
```

нарный предикат в унарный, привязав его второй аргумент, мы используем привязку *bind2nd*. В нашем примере требуется использовать выражение

```
bind2nd(less<int>(), 100)
```

чтобы указать, что мы хотим считать только значения, меньшие 100. Следующая программа показывает, как это все работает:

```
// binder.cpp:   Использование адаптера bind2nd для подсчета,
//               сколько из элементов массива меньше, чем 100.
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{  int   a[10] = {800, 3, 4, 600, 5, 6, 800, 71, 100, 2},
         n = 0;
   n = count_if(a, a + 10, bind2nd(less<int>(), 100));
   cout << n << endl; // Вывод: 6
   return 0;
}
```

Для привязывания первого аргумента существует привязка *bind1st*. К примеру, заменим условие $x < 100$ эквивалентным условием

$$100 > x$$

Этого можно добиться, привязав первый операнд y выражения

$$y > x$$

к значению 100. Следовательно, программа *binder.cpp* выдаст тот же результат, если мы заменим вызов *count_if* на следующий:

```
n = count_if(a, a + 10, bind1st(greater<int>(), 100));
```

Мы не будем подробно обсуждать реализации адаптеров *bind1st* и *bind2nd*, потому что их определения сложнее, чем использование в программах.

6.3. Отрицатели

Программисты часто используют унарный оператор ! (не). Например, выражение

```
!(x < y)
```

эквивалентно

```
x >= y
```

Подобным же образом на функции двух аргументов действует отрицатель

*not*2. Отрицатель является еще одной разновидностью *адаптера функции*; он реализуется в виде шаблонной функции, которая принимает в качестве аргумента объект, являющийся бинарным предикатом, например *less<int>*. Мы, следовательно, можем написать

```
not2(less<int>())
```

вместо

```
greater_equal<int>
```

Это выражение используется в следующей программе. Она сортирует массив из пяти элементов в нисходящем порядке:

```
// not2demo.cpp: Пример использования адаптера not2.
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{  int a[5] = {50, 30, 10, 40, 20};
   sort(a, a+5, not2(less<int>()));
   for (int i=0; i<5; i++) cout << a[i] << " ";
   // Вывод: 50 40 30 20 10
   cout << endl;
   return 0;
}
```

Приведенный выше вызов алгоритма *sort* можно заменить на

```
sort(a, a+5, greater_equal<int>());
```

или просто

```
sort(a, a+5, greater<int>());
```

Адаптер функции *not*1 изменяет унарные предикаты. Так как адаптеры *bind*1*st* и *bind*2*nd* возвращают унарный предикат, выражение типа

```
bind2nd(less<int>(), 100)
```

является приемлемым аргументом для адаптера *not*1, как показывает следующая программа:

```
// not1demo.cpp: Пример использования адаптера not1.
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;
```

```
int main()
{  int   a[10] = {800, 3, 4, 600, 5, 6, 800, 71, 100, 2},
         n = 0;
   // Подсчитаем, сколько имеется элементов не меньше 100:
   n = count_if(a, a + 10,
      not1(bind2nd(less<int>(), 100)));
   cout << n << endl; // Вывод: 4
   return 0;
}
```

Отметим, что подсчет числа элементов массива, равных или превышающих 100, может быть произведен с помощью более простого вызова алгоритма *count_if*:

```
n = count_if(a, a + 10,
   bind2nd(greater_equal<int>(), 100));
```

В любом случае будет выведено число 4, поскольку ровно четыре элемента (800, 600, 800, 100) массива *a* не меньше 100.

6.4. Два полезных базовых класса STL

Могли ли мы найти более простой пример для обсуждения адаптера *not*2, чем

```
sort(a, a+5, not2(less<int>()));
```

встречающийся в программе *not2demo.cpp* в предыдущем разделе? В частности, можем ли мы применить *not*2 к написанному нами функциональному объекту, например, таким образом:

```
sort(a, a+5, not2(iLessThen()));
```

если *iLessThen* — функциональный класс, определенный как

```
struct iLessThen {                  // ???
bool operator()(int x, int y)const {return x < y;}
};
```

Однако нам не удастся откомпилировать этот код. Адаптер *not*2 требует, чтобы его параметр шаблона был классом, производным от шаблонного класса *binary_function*, который упоминался в разделе 6.1. Этого легко добиться, добавив

```
: binary_function<int, int, bool>
```

сразу после имени класса *iLessThen* в приведенном выше объявлении этого класса. Три параметра шаблона *int*, *int* и *bool* соответствуют типам

двух аргументов и результата, которые можно было бы использовать при объявлении обычной функции для тех же целей, такой как

```
bool lessthan(int x, int y) {return x < y;}
```

Используя *binary_function<int, int, bool>* в качестве базового класса для *iLessThen*, можно написать следующую программу, эквивалентную программе *not2demo.cpp* из предыдущего раздела:

```
// not2own.cpp: Использование адаптера not2 с нашим
//              собственным функциональным объектом.
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

struct iLessThan: binary_function<int, int, bool> {
bool operator()(int x, int y)const {return x < y;}
};

int main()
{  int a[5] = {50, 30, 10, 40, 20};
   sort(a, a+5, not2(iLessThan()));
   for (int i=0; i<5; i++) cout << a[i] << " ";
   // Вывод: 50 40 30 20 10
   cout << endl;
   return 0;
}
```

Адаптер *not*1 также можно использовать с нашим функциональным объектом. В предыдущем разделе был использован *not*1 в выражении

```
not1(bind2nd(less<int>(), 100))
```

из программы *not*1*demo.cpp,* чтобы сосчитать, сколько элементов массива не меньше 100. В этом выражении мы заменим

```
bind2nd(less<int>(), 100)
```

более простой формой

```
LessThan100()
```

где *LessThan*100 – определенный нами класс, как показано в следующей программе:

```
// not1own.cpp: Использование адаптера not1 с нашим
//              собственным функциональным объектом.

#include <iostream>
#include <algorithm>
#include <functional>
```

```
using namespace std;

struct LessThan100: unary_function<int, bool> {
bool operator()(int x)const {return x < 100;}
};

int main()
{  int   a[10] = {800, 3, 4, 600, 5, 6, 800, 71, 100, 2},
      n = 0;
   // Подсчитаем количество элементов, не меньших 100:
   n = count_if(a, a+10, not1(LessThan100()));
   cout << n << endl; // Вывод: 4
   return 0;
}
```

В этой программе фрагмент

```
: unary_function<int, bool>
```

определяет базовый класс, наследником которого является класс *LessThan*100. Мы пишем *unary_function<int, bool>*, потому что адаптер *not*1 работает с унарным предикатом, который принимает в качестве параметра целое значение и возвращает значение типа *bool*. Напомним, что определение класса *unary_function* было приведено в разделе 6.1.

6.5. Функциональные объекты и алгоритм *transform*

Как видно из раздела 2.4, STL определяет следующие шаблонные классы, которые мы можем использовать как функциональные объекты, сопроводив их парой скобок:

```
plus<T>             minus<T>
multiplies<T>       divides<T>          modulus<T>
equal_to<T>         not_equal_to<T>
greater<T>          less<T>
greater_equal<T>    less_equal<T>
logical_and<T>      logical_or<T>
legate<T>           logical_not<T>
```

Чтобы проиллюстрировать работу некоторых функциональных объектов этого списка (не рассмотренных ранее), для начала обсудим алгоритмы *transform*. Этих алгоритмов в STL два – для унарной и для бинарной операций. Они нужны для преобразования всех элементов в диапазоне. Предположим, например, что нам требуется использовать элементы $a[i]$ целочисленного массива a для присваивания значений $b[i] = -a[i]$ другому массиву b. Один из способов достичь этого состоит в применении алгорит-

ма *transform* для унарной операции совместно с шаблоном *negate<T>*, как показывает следующая программа:

```
// negate.cpp: Алгоритм transform и negate<T>.
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{  int a[5] = {10, 20, -18, 40, 50}, b[5];
   transform(a, a + 5, b, negate<int>());
   for (int i=0; i<5; i++) cout << b[i] << "  ";
   // Вывод: -10  -20  18  -40  -50
   cout << endl;
   return 0;
}
```

Функциональный объект *logical_not<T>*() очень похож на *negate<T>*(). Если мы напишем

```
transform(a, a + 5, b, logical_not<int>());
```

массиву *b* будут присвоены значения $b[i] = !\,a[i]$. Напомним, что мы можем вместо *bool*, *true* и *false* писать *int*, 1 и 0. А в качестве источника и приемника допустимо использовать один и тот же массив. Например, вызов

```
transform(a, a + 5, a, negate<int>());
```

будет равнозначен умножению пяти элементов массива *a* на –1.

Перейдем теперь к знакомству с бинарной версией алгоритма *transform*. Этот вариант использует два контейнера-источника вместо одного, а также бинарный функциональный объект. Для примера предположим, что мы хотим использовать массивы *a* и *b* для вычисления следующей суммы в массиве *s*:

$$s[i] = a[i] + b[i]$$

Следующая программа показывает, как этого можно достичь с помощью шаблона *plus<int>*:

```
// plus.cpp: Алгоритм transform и plus<T>.
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{  int   a[5] = {10, 20, -18, 40, 50},
```

```
      b[5] = { 2,  2,   5,  3,  1}, s[5];
   transform(a, a + 5, b, s, plus<int>());
   for (int i=0; i<5; i++) cout << s[i] << "  ";
   // Вывод: 12  22  -13  43  51
   cout << endl;
   return 0;
}
```

Если необходимо, можно вместо отдельного массива *s* при вызове *transform* использовать один из источников *a* или *b*.

Точно так же, как шаблон *plus* применяется для операции +), следующие бинарные функциональные объекты *minus*, *multiplies*, *divides* и *modulus* используются для операций –, *, / и % соответственно. То же самое относится к бинарным функциональным объектам *equal_to*, *not_equal_to*, *greater*, *less*, *greater_equal*, *less_equal*, *logical_and* и *logical_or*, которые соответствуют операциям ==, !=, >, <, >=, <=, && и ||, возвращающим булевское значение. Два последних функциональных объекта (*logical_and* и *logical_or*) также обычно применяются к аргументам типа *bool*. Хотя этот список выглядит внушительно, нам часто для функции *transform* требуются операции, которые в нем отсутствуют. Тогда мы можем использовать свои собственные функциональные объекты, определив класс, который наследует от шаблона *binary_function* либо *unary_function*, как было показано в предыдущем разделе. Например, предположим, что нам опять даны два массива целых чисел *a* и *b* и мы хотим вычислить значения массива *result* таким образом:

```
result[i] = a[i] + 2 * b[i];
```

(скажем, для значений $i = 0, 1, ..., 4$). Следующая программа показывает, как решить эту задачу с помощью специально написанного класса *compute*:

```
// compute.cpp: Алгоритм transform
//                  и наш собственный функциональный объект.
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

struct compute: binary_function<int, int, int> {
   int operator()(int x, int y)const{return x + 2 * y;}
};

int main()
{  int   a[5] = {10, 20, -18, 40, 50},
         b[5] = { 2,  2,   5,  3,  1}, result[5];
   transform(a, a + 5, b, result, compute());
   for (int i=0; i<5; i++) cout << result[i] << "  ";
```

типов итераторов из тех, что уже были рассмотрены, а также некоторые другие типы.

Итераторы вставки

Как видно из программы *copy2.cpp* раздела 1.6, алгоритмы наподобие *copy* будут работать в режиме вставки, если мы напишем, например:

```
copy(v.begin(), v.end(), inserter(L, i));
```

Итератор вставки *inserter* является общим в том смысле, что мы указываем позицию, в данном примере *i*, в которой будет происходить вставка элементов. Если вставка в контейнер *L* должна происходить в конце, мы можем написать *L.end*(), как в следующей программе:

```
// copy3.cpp: Копирование вектора с помощью итератора
//            вставки.
#include <iostream>
#include <vector>
#include <list>
using namespace std;

int main()
{  int a[4] = {10, 20, 30, 40};
   vector<int> v(a, a+4);
   list<int> L(2, 123);
   copy(v.begin(), v.end(), inserter(L, L.end()));
   list<int>::iterator i;
   for (i=L.begin(); i != L.end(); ++i)
      cout << *i << " "; // Вывод: 123 123 10 20 30 40
   cout << endl;
   return 0;
}
```

Поскольку вставка в конце контейнера является часто встречающейся операцией, для нее существует специальный итератор вставки, называемый *back_inserter*. Приведенная выше программа работает точно так же, если мы заменим вызов алгоритма *copy* на вызов:

```
copy(v.begin(), v.end(), back_inserter(L));
```

Так как *back_inserter* всегда добавляет элементы в конец, он требует в качестве единственного аргумента контейнер.

Еще один итератор вставки, *front_inserter*, работает специфическим образом: каждый вставляемый элемент помещается в начале контейнера, что приводит к тому, что значения следуют в обратном порядке. Например, если мы заменим вызов алгоритма *copy* в программе *copy3.cpp* строчкой

```
copy(v.begin(), v.end(), front_inserter(L));
```

программа выведет следующие значения:

```
40 30 20 10 123 123
```

До сих пор мы применяли итераторы вставки только в качестве аргументов алгоритмов типа *copy* и *merge*. Мы можем использовать их другими способами. Например, вместо

```
L.push_front(111); L.push_back(999);
```

можно написать

```
*front_inserter(L)=111; *back_inserter(L)=999;
```

Обратные итераторы

Как мы видели в разделе 3.1, для типа *vector<int>* определены следующие типы итераторов, и легко догадаться, что подобные итераторы имеются и для других типов, например *list<double>*:

```
vector<int>::iterator
vector<int>::reverse_iterator
vector<int>::const_iterator
vector<int>::const_reverse_iterator
```

В разделе 1.2 обратный итератор использовался в следующем фрагменте для вывода всех элементов вектора *v* в обратном порядке:

```
vector<int>::reverse_iterator i;
for (i=v.rbegin; i != v.rend(); ++i)
   cout << * i << " ";
```

Типы итераторов, имена которых начинаются с *const*, нужны, когда контейнер сам имеет атрибут *const*, как в следующей программе, где функция *showlist* является вариантом функции с тем же именем из раздела 1.3:

```
// c_iter.cpp: const_iterator и
//             const_reverse_iterator
#include <iostream>
#include <list>
using namespace std;

void showlist(const list<int> &x)
{  // Вперед:
   list<int>::const_iterator i;
   for (i=x.begin(); i != x.end(); ++i)
      cout << *i << " ";
   cout << endl;   //  Вывод: 10 20 30
```

Поскольку здесь из всей STL мы используем только шаблон *ostream_iterator*, программа будет работать, если напишем *#include <iterator>* вместо *#include <vector>*.

С вводом дела обстоят несколько сложнее, потому что требуется обнаружить конец входного потока. Вспомним обсуждение конца файла в разделе 1.9. Ниже приведено нетривиальное решение задачи, которая заключается в том, чтобы прочитать все числа из файла *num.txt*, содержащего только целые числа в обычном формате.

```
// initer.cpp: Потоковый итератор ввода; чтение файла
//             с помощью операторов присваивания.
#include <iostream>
#include <fstream>
#include <vector>
#include <iterator>
using namespace std;

int main()
{  ifstream file("num.txt");
   if (file)
   {  istream_iterator<int> i(file), eof;
      int x;
      while (i != eof)
      {  x = *i++;
         cout << x << " ";
      }
   }  else cout << "Cannot open file num.txt.";
   cout << endl;
   return 0;
}
```

Например, если файл *num.txt* содержит две строчки

```
10 20
30
```

программа выведет на экран:

```
10 20 30
```

```
ForwardIterator1  find_first_of
   (ForwardIterator1  first1, ForwardIterator1  last1,
    ForwardIterator2  first2, ForwardIterator2  last2);
ForwardIterator1  find_first_of
   (ForwardIterator1  first1, ForwardIterator1  last1,
    ForwardIterator2  first2, ForwardIterator2  last2,
    BinaryPredicate pred);
ForwardIterator1  find_end
   (ForwardIterator1  first1, ForwardIterator1  last1,
    ForwardIterator2  first2, ForwardIterator2  last2);
ForwardIterator1  find_end
   (ForwardIterator1  first1, ForwardIterator1  last1,
    ForwardIterator2  first2, ForwardIterator2  last2,
    BinaryPredicate pred);
```

Мы не будем повторять предыдущее обсуждение алгоритмов *find*, *find_if*, *for_each*, *count* и *count_if*.

Алгоритм *find_first_of* похож на *find*, но вместо поиска элемента с заданным значением он ищет в первом диапазоне элемент, значение которого равно какому-либо элементу из второго диапазона. Если такие элементы присутствуют в диапазоне, он возвращает итератор, ссылающийся на первый из них. Алгоритм *find_end*, напротив, возвращает итератор, указывающий на начало последнего полного вхождения второго диапазона в первый. (Если нужно найти первое вхождение, используйте алгоритм *search*, который рассмотрен далее в разделе 7.1.5.) В случае неудачи *find_first_of* и *find_end* возвращают *last*1. Следующая программа демонстрирует работу обоих алгоритмов:

```
// find_end: Алгоритмы find_first_of и find_end.
#include <iostream>
#include <algorithm>

using namespace std;
int main()
{  int   a[10] = {3, 2, 5, 7, 5, 8, 7, 5, 8, 5},
         b[2] = {5, 8}, *p1, *p2;
   p1 = find_first_of(a, a+7, b, b+3);
   p2 = find_end(a, a+10, b, b+2);
   cout << p1 - a << " " << p2 - a << endl;
   return 0;
}
```

Вывод этой программы:

```
2 7
```

объясняется тем, что $a[2]$ (=5) является первым элементом a, который входит также и в b, а $a[7]$ (=5) является начальным элементом последней полной последовательности 5, 8 (заданной массивом b), которая встречается в a.

В STL определены также версии *find_first_of* и *find_end*, принимающие в качестве дополнительного параметра для проведения сравнений бинарный предикат. Мы можем использовать этот предикат для замены проверки на равенство другим условием; подробнее применение этого предиката рассмотрено в следующем разделе на примере алгоритма *adjacent_find*.

7.1.2. Алгоритм *adjacent_find*

```
ForwardIterator adjacent_find
   (ForwardIteratorfirst, ForwardIteratorlast,
    const T&  value);
ForwardIterator adjacent_find
   (ForwardIteratorfirst, ForwardIteratorlast,
    Predicate  pred);
```

Для поиска в последовательном контейнере определены два алгоритма *adjacent_find* (найти рядом). Первый ищет соседние элементы $a[k]$ и $a[k+1]$, которые равны друг другу; второй ищет соседние элементы, которые удовлетворяют заданному условию. В следующей программе показано, как работают оба варианта. Программа *adjacent.cpp* осуществляет поиск в массиве *a* четыре раза:

двух элементов, $a[k]$ и $a[k+1]$, которые равны между собой;
двух элементов, $a[k]$ и $a[k+1]$, удовлетворяющих условию $a[k] > a[k+1]$;
двух элементов, $a[k]$ и $a[k+1]$, удовлетворяющих условию $a[k]^2 = a[k+1]$;
двух элементов, $a[k]$ и $a[k+1]$, удовлетворяющих условию $a[k]^3 = a[k+1]$.

Если искомые соседние элементы не найдены, функция *adjacent_find* возвращает, как обычно, итератор, следующий за итератором, указывающим на последний элемент. Так, если бы мы вместо массива *a* использовали вектор *v*, это значение совпало бы со значением *v.end*(). Значения в массиве *a* подобраны таким образом, что первые три поиска происходят удачно, а последний – нет:

```
// adjacent.cpp: Алгоритмы adjacent_find.
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <functional>
using namespace std;

bool is_square(int x, int y){return x * x == y;}
bool is_cube(int x, int y){return x * x * x == y;}

int main()
{  int a[10] =
   {5, 10, 28, 20, 10, 5, 25, 10, 10, 90}, *p, k;
   for (k=0; k<10; k++) cout << setw(3) << k;
```

```
   cout << endl;
   for (k=0; k<10; k++) cout << setw(3) << a[k];
   cout << endl;
   p = adjacent_find(a, a+10);
   k = p - a;
   if (k != 10)
      cout  << "a[k] = a[k+1], found at k = " << k
            << endl;
   p = adjacent_find(a, a+10, greater<int>());
   k = p - a;
   if (k != 10)
      cout  << "a[k] > a[k+1], found at k = " << k
            << endl;
   p = adjacent_find(a, a+10, is_square);
   k = p - a;
   if (k != 10)
      cout  << "a[k]*a[k] == a[k+1], found at k = "
            << k << endl;

   p = adjacent_find(a, a+10, is_cube);
   k = p - a;
   if (k != 10)
      cout  << "a[k]*a[k]*a[k] == a[k+1], found at k = "
            << k << endl;
   else cout   << "If not found: k = " << k << endl;
   return 0;
}
```

Программа выведет:

```
   0  1  2  3  4  5  6  7  8  9
   5 10 28 20 10  5 25 10 10 90
a[k] = a[k+1], found at k = 7
a[k] > a[k+1], found at k = 2
a[k]*a[k] == a[k+1], found at k = 5
If not found: k = 10
```

Предикаты в этой программе записаны как функции. Из предыдущего обсуждения (см., например, главу 6) очевидно, что мы можем использовать вместо функций функциональные объекты. Например, мы можем заменить функцию *is_square* следующим определением класса:

```
struct sq{
bool operator()(int x, int y){return x * x == y;}
};
```

если также заменим

```
p = adjacent_find(a, a+10, is_square);
```

на

```
p = adjacent_find(a, a+10, sq());
```

Этот подход применим и к другим предикатам в настоящей главе.

7.1.3. Отличие

```
pair<InputIterator1, InputIterator2> mismatch
   (InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2);
pair<InputIterator1, InputIterator2> mismatch
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, BinaryPredicate binary_pred);
```

Существуют два варианта алгоритма *mismatch*. Первый ищет одновременно в двух последовательных контейнерах несовпадающие элементы, стоящие в одной позиции. Другими словами, если мы обозначим эти элементы как x и y, будет выполнено условие $!(x == y)$. Второй вариант использует передаваемый дополнительно предикат вместо оператора ==. Следующая программа демонстрирует оба варианта:

```
// mismatch.cpp: Алгоритмы mismatch.
#include <iostream>
#include <iomanip>
#include <algorithm>
using namespace std;

bool smalldif(int a, int b){return abs(a - b) < 25;}

int main()
{  int   a[4] = {50, 80, 30, 90},
      b[5] = {50, 80, 10, 40, 20}, *pa, *pb, k;
   pair<int*, int*> difpos(0, 0);
   for (k=0; k<5; k++) cout << setw(3) << k;
   cout << endl;
   for (k=0; k<4; k++) cout << setw(3) << a[k];
   cout << endl;
   for (k=0; k<5; k++) cout << setw(3) << b[k];
   cout << endl;
   difpos = mismatch(a, a+4, b);
   pa = difpos.first;
   pb = difpos.second;
   cout  << "Different elements "
         << *pa << " and " << *pb
         << " found at position " << pa - a << endl;
   difpos = mismatch(a, a+4, b, smalldif);
   pa = difpos.first;
   pb = difpos.second;
```

```
   cout  << "Difference of at least 25 found between\n"
         << *pa << " and " << *pb
         << " at position " << pa - a << endl;
   return 0;
}
```

В обоих случаях отличия находятся, как видно из вывода этой программы:

```
  0  1  2  3  4
 50 80 30 90
 50 80 10 40 20
Different elements 30 and 10 found at position 2
Difference of at least 25 found between
90 and 40 at position 3
```

Обратите внимание, что вторая последовательность должна быть не короче первой. Если бы отличий найти не удалось (например, при замене в тексте программы 25 на 250), возвращаемая пара итераторов соответствовала бы позиции 4 во входных массивах.

7.1.4. Сравнение на равенство

```
bool equal
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2);
bool equal
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, BinaryPredicate binary_pred);
```

Алгоритмов *equal* также два. Первый определяет равенство двух последовательностей (в указанных диапазонах); второй работает аналогично первому, но использует передаваемый дополнительным параметром бинарный предикат вместо оператора сравнения на равенство ==. Нижеприведенная программа демонстрирует обе версии:

```
// equal.cpp: Алгоритмы equal.
#include <iostream>
#include <iomanip>
#include <algorithm>
using namespace std;

bool approx(int a, int b){return abs(a - b) <= 1;}

int main()
{  int   a[4] = {50, 80, 30, 90},
         b[5] = {50, 80, 30, 90, 20},
         c[4] = {50, 79, 30, 90};
   cout << equal(a, a+4, b) << endl;         // 1
```

```
    cout << equal(a, a+4, c) << endl;          // 0
    cout << equal(a, a+4, c, approx) << endl; // 1
    return 0;
}
```

Функция возвращает булевское значение *true* (= 1) или *false* (= 0), как показано в комментариях. Поскольку четыре элемента *a* совпадают с первыми четырьмя элементами *b*, первый вызов *equal* возвращает *true*. Второй вызов возвращает *false*, потому что *a*[1] = 80, в то время как *c*[1] = 79. В третьем вызове разница 80 – 79 является достаточно малой (в соответствии с определением функции *approx*), чтобы функция *equal* вернула *true*. Обратите внимание, что функция *equal* не возвращает информацию о позиции, в которой последовательности не совпадают. Если нужна такая информация, следует пользоваться алгоритмом *mismatch*.

7.1.5. Поиск подпоследовательности

```
ForwardIterator1 search
   (ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);
ForwardIterator1 search
   (ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2,
    BinaryPredicate  predicate) ;
```

Первый вариант алгоритма *search* является обобщением известной функции поиска из библиотеки C, которая используется в следующем фрагменте:

```
#include <string.h>
...
char *p = strstr("ABCDEF", "CD"); /* *p == 'C' */
```

Библиотечная функция *strstr* возвращает *NULL*, если второй аргумент не является подстрокой первого.

Первый вариант *search* используется для поиска (меньшей) подпоследовательности в (большей) последовательности, использует два первых параметра для задания большей последовательности (в которой ищут) и два следующих для меньшей (которую ищут). Второй вариант использует передаваемый пятым параметром бинарный предикат вместо оператора сравнения на равенство ==. Следующая программа показывает работу обоих алгоритмов *search*:

```
// search.cpp: Алгоритмы search.
#include <iostream>
#include <algorithm>
using namespace std;
```

```
// transfor.cpp: Два алгоритма transform.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int plus1(int x){return x + 1;}
int largersq(int x, int y){return x > y ? x*x : y*y;}

int main()
{  int   a[5] = {2, -4, 3, 5, 1},
         b[5] = {1, -3, 5, 2, 4};
   vector<int> result;
   transform(a, a + 5,
      inserter(result, result.begin()), plus1);
   copy(result.begin(), result.end(),
      ostream_iterator<int>(cout, " "));
   cout << endl;  // Вывод: 3 -3 4 6 2
   transform(a, a+5, b, result.begin(), largersq);
   copy(result.begin(), result.end(),
      ostream_iterator<int>(cout, " "));
   cout << endl;  // Вывод: 4 9 25 25 16
   return 0;
}
```

Как указано в комментариях, первый результат получается добавлением единицы ко всем элементам массива *a*. Второй вызов *transform* вычисляет квадрат большего из соответствующих элементов массивов *a* и *b*. К примеру, *result*[0] = 4, потому что это квадрат большего значения (2) из *a*[0] и *b*[0].

7.2.2. Копировать

```
OutputIterator copy
   (InputIterator first, InputIterator last,
    OutputIterator result);
BidirectionalIterator2 copy_backward
   (BidirectionalIterator1 first1,
    BidirectionalIterator1 last1,
    BidirectionalIterator2 last2);
```

Алгоритм *copy* рассматривался в разделе 1.6. В примерах этого раздела источник и приемник копирования никогда не перекрывались. Если же они перекрываются, необходимо правильно выбрать, какой из алгоритмов использовать: *copy*, который копирует элементы в прямом порядке, либо *copy_backward*, копирующий в обратном порядке. Например, если мы хотим сдвинуть элементы *a*[1], *a*[2] и *a*[3] массива *a* на одну позицию влево, самый элементарный способ достичь этого следующий:

```
for (i=1; i<4; i++) a[i-1] = a[i];
```

Алгоритм *copy* делает то же самое, как показывает следующая программа:

```
// copy1.cpp: Сдвиг элементов массива влево.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{  int a[4] = {10, 20, 30, 40}, i;
   cout << "Before shifting left: ";
   for (i=0; i<4; i++) cout << a[i] << " " ;
   copy(a+1, a+4, a);
   cout << "\nAfter shifting left:  ";
   for (i=0; i<4; i++) cout << a[i] << " " ;
   cout << endl;
   return 0;
}
```

Напомним, что первые два параметра *copy* задают источник, а третий – приемник. Программа сдвигает три элемента массива влево, как видно из ее вывода:

```
Before shifting left: 10 20 30 40
After shifting left:  20 30 40 40
```

Противоположная операция – сдвиг элементов $a[0]$, $a[1]$ и $a[2]$ массива, содержащего четыре элемента, вправо – может быть проведена следующим образом:

```
for (i=2; i>=0; i--) a[i+1] = a[i];
```

Обратите внимание, что мы должны двигаться назад, уменьшая *i*. Программа, приведенная ниже, показывает, что алгоритм *copy_backward* делает то же самое:

```
// copy2.cpp: Сдвиг элементов массива вправо.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{  int a[4] = {10, 20, 30, 40}, i;
   cout << "Before shifting right: ";
   for (i=0; i<4; i++) cout << a[i] << " " ;
   copy_backward(a, a+3, a+4);
   cout << "\nAfter shifting right:  ";
   for (i=0; i<4; i++) cout << a[i] << " " ;
```

```
    cout << endl;
    return 0;
}
```

Как и для *copy*, первые два аргумента *copy_backward* указывают источник. Однако третий аргумент указывает позицию за последним элементом приемника; другими словами, это значение будет уменьшаться *перед* каждым шагом алгоритма, включая первый шаг. Вывод этой программы следующий:

```
Before shifting right: 10 20 30 40
After shifting right:  10 10 20 30
```

В рассмотренных программах несколько раз встречается строчка:

```
for (i=0; i<4; i++) cout << a[i] << " " ;
```

Вместо нее мы можем использовать тот же алгоритм *copy*, как рассказано ранее в разделе 1.9:

```
copy(a, a+4, ostream_iterator<int>(cout, " "));
```

В некоторых дальнейших примерах нами будет использован также и этот метод.

7.2.3. Переместить по кругу

```
void rotate
    (ForwardIterator first, ForwardIterator middle,
     ForwardIterator last);
OutputIterator rotate_copy
    (ForwardIterator first, ForwardIterator middle,
     ForwardIterator last, OutputIterator result);
```

Иногда возникает необходимость вместо сдвига на несколько позиций влево или вправо переместить элементы последовательного контейнера по кругу. Например, преобразовать

```
10 20 30 40
```

в следующую последовательность

```
20 30 40 10
```

Это преобразование может считаться циклическим сдвигом влево на одну позицию. Однако с таким же успехом мы можем считать его циклическим сдвигом вправо на три позиции. Поэтому существует только один алгоритм *rotate*. Вместо того чтобы указывать количество позиций и направление

сдвига, мы просто задаем элемент, который должен стать первым. Аргумент функции, используемый для этой цели, следует вторым, а первый и третий аргументы определяют диапазон, к которому применяется циклический сдвиг. Чтобы осуществить сдвиг в приведенном выше примере для массива из четырех элементов, требуется, следовательно, вызвать алгоритм *rotate* следующим образом:

```
rotate(a, a+1, a+4);
```

Параметр $a + 1$ показывает, что элемент $a[1] = 20$ должен стать первым в новой последовательности. Целиком программа приведена ниже:

```
// rotate.cpp: Циклический сдвиг.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{  int a[4] = {10, 20, 30, 40};
   cout << "Initial contents of array a: ";
   copy(a, a+4, ostream_iterator<int>(cout, " "));
   rotate(a, a+1, a+4);
   cout << "\nAfter rotate(a, a+1, a+4):   ";
   copy(a, a+4, ostream_iterator<int>(cout, " "));
   cout << endl;
   return 0;
}
```

Программа выводит следующий текст:

```
Initial contents of array a: 10 20 30 40
After rotate(a, a+1, a+4):   20 30 40 10
```

Если после имеющегося в программе вызова *rotate* добавить следующий вызов:

```
rotate(a, a+3 a+4);
```

элемент $a[3] = 10$ снова станет $a[0]$, что восстановит в результате первоначальную последовательность 10, 20, 30, 40.

В STL определена также функция *rotate_copy*, которая помещает результат циклического сдвига в другой контейнер, так что источник остается неизменным. Контейнер-приемник задается четвертым аргументом функции, как в следующей программе:

```
// rotcopy.cpp: Алгоритм rotate_copy.
#include <iostream>
#include <algorithm>
```

```
using namespace std;

int main()
{  int a[4] = {10, 20, 30, 40}, b[4];
   rotate_copy(a, a+1, a+4, b);
   copy(b, b+4, ostream_iterator<int>(cout, " "));
   cout << endl; // Вывод: 20 30 40 10
   return 0;
}
```

7.2.4. Обменять

```
void swap
   (T& x, T& y);
void iter_swap
   (ForwardIterator1& a, ForwardIterator1& b);
ForwardIterator2 swap_ranges
   (ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2);
```

Алгоритм *swap* меняет значения двух объектов одного и того же типа, как показывает следующая программа:

```
// swap.cpp: Поменять два значения.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{  double   a = 3.14159,
            b = 2.71828;
   swap(a, b);
   cout << a << " " << b << endl;
   //   2.71828     3.14159
   return 0;
}
```

Если мы имеем два итератора, то можем использовать алгоритм *iter_swap* для обмена местами значений, на которые ссылаются эти итераторы, например:

```
// it_swap.cpp: Алгоритм iter_swap.
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{  list<int> L;
```

```
   list<int>::iterator i, j;
   L.push_back(123);
   L.push_back(456);
   copy(L.begin(), L.end(),
      ostream_iterator<int>(cout, " "));
   cout << endl;  // Вывод: 123 456
   i = L.begin();
   j = i;
   ++j;
   iter_swap(i, j);
   copy(L.begin(), L.end(),
      ostream_iterator<int>(cout, " "));
   cout << endl;  // Вывод: 456 123
   return 0;
}
```

Алгоритм *swap_ranges* меняет местами два диапазона значений. Эти контейнеры не должны перекрываться, и они могут относиться к разным типам, как показывает следующая программа:

```
// swranges.cpp: Поменять местами диапазоны значений.
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{  int a[3] = {10, 20, 30};
   vector<int> v;
   v.push_back(100); v.push_back(200); v.push_back(300);
   swap_ranges(v.begin(), v.end(), a);
   cout << "After swap_ranges:\n";
   cout << "i   a[i]   v[i]\n";
   for (int i=0; i<3; i++)
      cout  << i << "    " << a[i] << "      "
            << v[i] << endl;
   return 0;
}
```

Вывод программы:

```
After swap_ranges:
i   a[i]   v[i]
0   100     10
1   200     20
2   300     30
```

Как видно из рассмотренного примера программы, первые два аргумента функции *swap_ranges* задают один диапазон, а третий аргумент задает

начало второго диапазона. Поскольку операция обмена симметрична по отношению к своим аргументам, программа будет работать точно так же, если мы заменим вызов

```
swap_ranges(v.begin(), v.end(), a);
```

на

```
swap_ranges(a, a+3, v.begin());
```

7.2.5. Заменить

```
void replace
   (ForwardIterator first, ForwardIterator last,
    const T& old_value, const T& new_value);
void replace_if
   (ForwardIterator first, ForwardIterator last,
    Predicate pred, const T& new_value);
OutputIterator replace_copy
   (InputIterator first, InputIterator last,
    OutputIterator result,
    const T& old_value, const T& new_value);
OutputIterator replace_copy_if
   (InputIterator first, InputIterator last,
    OutputIterator result,
    Predicate pred, const T& new_value);
```

В разделе 1.10 нами уже обсуждался алгоритм *replace*. Напомним, что мы можем в имеющемся целочисленном массиве *a*, определенном как

```
int a[5] = {8, 8, 8, 0, 8};
```

заменить все элементы со значением 8 на 1, выполнив

```
replace(a, a+5, 8, 1);
```

что даст в результате массив, содержащий 1, 1, 1, 0, 1.

В STL также определена функция *replace_if*, которая имеет отличающийся третий аргумент – унарный предикат, задающий некоторое условие. Все элементы, удовлетворяющие этому условию, будут заменены на значение, заданное четвертым аргументом функции. Алгоритм *replace_if* демонстрирует следующая программа:

```
// repl_if.cpp: Заменить все ненулевые элементы массива
//              на 1.
#include <iostream>
#include <algorithm>
using namespace std;

bool nonzero(int x) {return x != 0;}
```

```
int main()
{  int a[5] = {10, 20, 30, 0, 40};
   replace_if(a, a+5, nonzero, 1);
   cout << "After replace_if:\n";
   copy(a, a+5, ostream_iterator<int>(cout, " "));
   cout << endl;
   return 0;
}
```

Все ненулевые элементы в массиве {10, 20, 30, 0, 40} заменяются на 1, поэтому программа выводит:

```
After replace_if:
1 1 1 0 1
```

Два алгоритма *replace_copy* и *replace_copy_if* очень похожи на *replace* и *replace_if*. Вместо того чтобы производить замену в исходном контейнере, они сначала копируют этот контейнер в другой, а затем заменяют элементы в новом контейнере, не модифицируя источник. Третий аргумент этих функций задает начало контейнера, в котором будет храниться результат копирования и замены, что мы покажем в следующей программе:

```
// replcopy.cpp: replace_copy и replace_copy_if.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool not_ten(int x) {return x != 10;}

int main()
{  int a[5] = {10, 20, 20, 10, 40};
   cout << "Array a:\n";
   copy(a, a+5, ostream_iterator<int>(cout, " "));
   vector<int> v;
   vector<int>::iterator i;
   replace_copy(a, a+5, inserter(v, v.begin()),
      20, 49);
   cout << "\n\nAfter the execution of\n   replace_copy"
      "(a, a+5, inserter(v, v.begin()), 20, 49);"
      "\nvector v has the following contents:\n";
   copy(v.begin(), v.end(),
      ostream_iterator<int>(cout, " "));
   replace_copy_if(a, a+5, v.begin(), not_ten, 99);
   cout << "\n\nAfter the execution of\n"
      "   replace_if_copy(a, a+5, v.begin(), "
      "not_ten, 99);"
      "\nvector v has the following contents:\n";
```

7.2.8. Породить

```
void generate
   (ForwardIterator first, ForwardIterator last,
    Generator gen);
void generate_n
   (OutputIterator first, Size n,
    Generator gen);
```

Вместо того чтобы использовать постоянное значение для заполнения контейнера, иногда желательно вычислить свое значение для каждого элемента. Это можно сделать с помощью алгоритма *generate* (породить). В качестве третьего параметра этого алгоритма выступает функция или функциональный объект. Следующая программа помещает значения 10, 12, 14, 16 и 18 в массив *a*:

```
// generate.cpp: Алгоритм generate.
#include <iostream>
#include <algorithm>
using namespace std;

struct funobj {
   int i;
   funobj(): i(8){}
   int operator()(){return i += 2;}
};

int main()
{  int a[5];
   generate(a, a+5, funobj());
   copy(a, a+5, ostream_iterator<int>(cout, " "));
   cout << endl;  // Вывод: 10 12 14 16 18
   return 0;
}
```

Обратите внимание, что мы используем функциональный объект, который содержит кроме определения функции *operator*() также определение конструктора и переменную-член *i* целого типа. Эта переменная при инициализации объекта получает значение 8 и увеличивает свое значение на 2 каждый раз, когда происходит вызов функции *operator*(), вследствие чего эта функция последовательно возвращает значения 10, 12, 14, 16 и 18.

Мы могли бы заполнить массив *a* теми же значениями, используя функцию

```
int fun()
{  static int i=8;
   return i += 2;
}
```

вместо класса *funobj* и изменив вызов *generate* следующим образом:

```
generate(a, a+5, fun);
```

Однако имеется тонкое различие: если бы мы вызывали *generate* в программе два раза и использовали функцию *fun* вместо функционального объекта, то во втором вызове *generate* переменная *i* не получит значение 8; когда же мы используем функциональный объект *funobj*, каждый вызов алгоритма *generate* будет начинаться с присваивания 8 этой переменной.

Если *a* определен как вектор или двусторонняя очередь, содержащая, по крайней мере, 5 элементов, мы можем использовать вызов

```
generate(a.begin(), a.begin()+5, fun);
```

Однако со списком мы не можем поступить подобным образом, поскольку оператор сложения не определен для двунаправленных итераторов. К счастью, существует функция *generate_n*, которая работает в том числе и со списком. Следующая программа отличается от предыдущей двумя аспектами: она использует функцию *fun* вместо функционального объекта *funobj*(), а также использует алгоритм *generate_n*, чтобы мы могли задать диапазон из пяти элементов для списка.

```
// gen_n.cpp: Алгоритм generate_n.
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;

int fun()
{  static int i=8;
   return i += 2;
}

int main()
{  list<int> a(5);
   generate_n(a.begin(), 5, fun);
   copy(a.begin(), a.end(),
      ostream_iterator<int>(cout, " "));
   cout << endl;  // Вывод: 10 12 14 16 18
   return 0;
}
```

7.2.9. Убрать повторы

```
ForwardIterator unique
   (ForwardIterator first, ForwardIterator last);
ForwardIterator unique
   (ForwardIterator first, ForwardIterator last,
    BinaryPredicate binary_pred);
```

```
OutputIterator unique_copy
   (InputIterator first, InputIterator last,
    OutputIterator result);
OutputIterator unique_copy
   (InputIterator first, InputIterator last,
    OutputIterator result,
    BinaryPredicate binary_pred);
```

Алгоритм *unique* удаляет стоящие рядом одинаковые элементы в заданном диапазоне. Этот алгоритм напоминает *remove* тем, что не изменяет размер контейнера. Размер массива мы все равно не можем изменить, но алгоритм не изменяет размеры и любых других контейнеров, будь то вектор, двусторонняя очередь или список. Кстати, если возникает необходимость использовать этот алгоритм для списков, лучше воспользоваться функцией-членом *unique* класса *list*, поскольку она более эффективна. Напомним, что эту функцию-член мы рассматривали в разделе 3.5. Алгоритм *unique* возвращает новый логический конец данных, как показывает следующая программа:

```
// unique1.cpp: Первый из алгоритмов unique.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{  int a[10] = {3, 4, 3, 3, 4, 4, 5, 3, 3, 3}, *p;
   p = unique(a, a+10);
   cout << "New logical contents of array a:\n";
   copy(a, p, ostream_iterator<int>(cout, " "));
   cout << endl;
   return 0;
}
```

Программа выводит следующий текст:

```
New logical contents of array a:
3 4 3 4 5 3
```

Вместо использования операции сравнения на равенство мы можем задать любой бинарный предикат, воспользовавшись вторым вариантом алгоритма с тем же именем, который принимает предикат в качестве третьего аргумента. Следующая программа использует этот вариант алгоритма *unique*, чтобы удалить каждый из элементов вектора, который больше предыдущего на единицу:

Существуют также две функции *unique_copy*, которые похожи на функции *unique*, за исключением того, что результат их работы копируется в другой контейнер. Мы могли бы использовать эти функции в программе *unique2.cpp* следующим образом. Фрагмент

```
vector<int> v;
vector<int>::iterator i;
for (int k=0; k<10; k++) v.push_back(a[k]);
i = unique(v.begin(), v.end(), onehigher);
```

можно заменить на более короткий

```
vector<int> v(10,0);
vector<int>::iterator i;
i = unique_copy(a, a+10, v.begin(), onehigher);
```

После такой замены программа будет работать так же, как и приведенная выше версия.

Другой вариант функции *unique_copy* похож на первую версию *unique*. Он проверяет последовательные элементы на равенство. Мы можем использовать эту версию в выражении

```
i = unique_copy(a, a+10, v.begin());
```

7.2.10. Расположить в обратном порядке

```
void reverse
   (BidirectionalIterator first,
    BidirectionalIterator last);
OutputIterator reverse_copy
   (BidirectionalIterator first,
    BidirectionalIterator last, OutputIterator result);
```

Напомним, что алгоритм *reverse* мы обсуждали в разделе 1.10. Алгоритм *reverse_copy* не изменяет заданный диапазон, а помещает элементы в обратном порядке в другой контейнер, как показано в следующей программе:

```
// rcopy.cpp: Алгоритм reverse_copy помещает
//            результат в другой контейнер.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{  int a[3] = {10, 20, 30}, b[3];
   reverse_copy(a, a+3, b);
   copy(b, b+3, ostream_iterator<int>(cout, " "));
   cout << endl; // Вывод: 30 20 10
   return 0;
}
```

рые этому условию не соответствуют. В следующей программе массив *a* содержит целые числа, часть из них меньше 50. Мы используем алгоритм *partition*, которому передаем предикат «меньше 50» для перестановки элементов массива *a* таким образом, что все элементы, удовлетворяющие указанному условию, будут предшествовать элементам, чье значение больше или равно 50. Алгоритм возвращает значение итератора, ссылающееся на первый элемент во втором разделе:

```
// partitio.cpp: Алгоритм partition.
#include <iostream>
#include <algorithm>
using namespace std;

bool below50(int x){return x < 50;}

int main()
{  int a[8] = {70, 40, 80, 20, 50, 60, 50, 10};
   int *p = partition(a, a+8, below50);
   copy(a, p, ostream_iterator<int>(cout, " "));
   cout << "  ";
   copy(p, a+8, ostream_iterator<int>(cout, " "));
   cout << endl;
   return 0;
}
```

Массив *a* содержит три элемента (40, 20 и 10), которые меньше, чем 50. После вызова *partition* эти элементы группируются в начале массива, как показывает вывод программы:

```
10 40 20   80 50 60 50 70
```

Значение $p - a$ равно длине (3) первого раздела. Один из двух разделов может оказаться пустым. Например, если в функции *below*50 мы заменим 50 на 1000, получим $p - a = 8$, а если в этой функции заменим 50 на 0, $p - a$ станет равным 0.

Обратите внимание, что элементы 40, 20 и 10, меньшие 50, не сохраняют свой первоначальный относительный порядок, как не сохраняют его и остальные элементы (70, 80, 50, 60, 50). Если мы хотим, чтобы относительный порядок элементов в каждом из разделов остался прежним, нам достаточно заменить вызов *partition* на *stable_partition*, написав

```
int *p = stable_partition(a, a+8, below50);
```

В результате мы получим следующий вывод программы, где элементы в каждом из разделов сохраняют тот же порядок, что и в исходной последовательности:

```
40 20 10   70 80 50 60 50
```

7.3.3. Стабильная сортировка

```
void stable_sort
   (RandomAccessIterator first,
    RandomAccessIterator last);
void stable_sort
   (RandomAccessIterator first,
    RandomAccessIterator last, Compare comp);
```

Чтобы обсудить концепцию стабильной сортировки, лучше сортировать структуры, а не целые числа. Следующая программа сортирует массив из 20 записей, каждая из которых содержит строку и целое число. В качестве ключей будут использованы строки, поэтому после сортировки они разместятся в лексикографическом порядке. Мы реализуем это, определяя оператор < таким образом, что $a[i] < a[j]$, если $a[i].s$ лексикографически предшествует $a[j].s$. До сортировки в переменных-членах *num* структур, хранящихся в элементах массива $a[0]$, $a[1]$, ..., $a[19]$, содержатся значения 10, 11, ..., 29.

```
// sortrec.cpp: Сортировка структур.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

struct rectype
{  string s; int num;
   bool operator<(const rectype &b)const
   {  return s < b.s;
   }
};

int main()
{  const int N = 20;
   string t[20] =
   {"Judy", "John", "John", "Judy", "John",
    "Judy", "Paul", "Judy", "Paul", "Mary",
    "Mary", "John", "Judy", "Paul", "John",
    "Paul", "Judy", "John", "Judy", "Judy"};
   int k;
   rectype a[N];
   for (k=0; k<N; k++)
   {  a[k].s = t[k]; a[k].num = 10 + k;
   }
   sort(a, a+N);
   for (k=0; k<N; k++)
   {  cout  << a[k].s << " "
            << a[k].num << (k % 5 == 4 ? "\n" : "   ");
```

```
    }
    return 0;
}
```

Программа создает следующий вывод:

```
John 11  John 12  John 14  John 27  John 24
John 21  Judy 29  Judy 28  Judy 26  Judy 22
Judy 17  Judy 15  Judy 13  Judy 10  Mary 20
Mary 19  Paul 18  Paul 23  Paul 25  Paul 16
```

В этом отсортированном списке имена следуют в лексикографическом порядке. Как вы можете заметить уже в первой из четырех выведенных строк, номера, связанные с одним и тем же ключом, следуют не по возрастанию. Поскольку до сортировки они шли в порядке возрастания, мы видим, что относительный порядок при сортировке не сохранен. Например, в сортированном массиве записи (*John*, 27) и (*John*, 24) появляются перечислением, хотя изначально запись (*John*, 24) была расположена раньше, чем (*John*, 27). Это происходит из-за того, что алгоритм *sort нестабилен*. Однако для него есть и стабильная версия. Чтобы использовать ее, мы просто заменим вызов алгоритма *sort* на следующий:

```
stable_sort(a, a+N);
```

После этой модификации программа напечатает:

```
John 11  John 12  John 14  John 21  John 24
John 27  Judy 10  Judy 13  Judy 15  Judy 17
Judy 22  Judy 26  Judy 28  Judy 29  Mary 19
Mary 20  Paul 16  Paul 18  Paul 23  Paul 25
```

Записи с равными ключами (как те, которые содержат имя *John*) теперь располагаются в порядке возрастания чисел; другими словами, относительный порядок этих записей сохраняется.

Как и для алгоритма *sort*, для *stable_sort* существует версия, которая в качестве третьего аргумента принимает предикат.

7.3.4. Частичная сортировка

```
void partial_sort
   (RandomAccessIterator first,
    RandomAccessIterator middle,
    RandomAccessIterator last);
void partial_sort
   (RandomAccessIterator first,
    RandomAccessIterator middle,
    RandomAccessIterator last, Compare comp);
```

```
RandomAccessIterator partial_sort_copy
   (InputIterator first, InputIterator last,
    RandomAccessIterator result_first,
    RandomAccessIterator result_last);
RandomAccessIterator partial_sort_copy
   (InputIterator first, InputIterator last,
    RandomAccessIterator result_first,
    RandomAccessIterator result_last, Compare comp);
```

Если нас интересуют только первые *n* элементов полностью отсортированной последовательности, где *n* меньше длины последовательности, то нет необходимости полностью сортировать эту последовательность. В следующей программе элементы *a*[0], *a*[1], *a*[2] и *a*[3] получают те же значения, которые они имели бы в полностью отсортированном массиве:

```
// partsort.cpp: Частичная сортировка.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{  int a[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
   partial_sort(a, a+4, a+10);
   for (int k=0; k<10; k++)
      cout << a[k] << (k == 3 ? "   " : " ");
   cout << endl;
   return 0;
}
```

Программа выводит следующую строку:

```
0 1 2 3   9 8 7 6 5 4
```

Элементы после первых четырех могут появляться в любом порядке; хотя в этом примере они располагаются в том же относительном порядке, в каком были изначально, мы не должны рассчитывать на такое поведение.

Другая версия алгоритма *partial_sort* принимает в качестве четвертого аргумента предикат для сравнения, как аналогичная версия алгоритма *sort*.

Если мы хотим разместить результаты в диапазоне, отличном от исходного, то можем использовать алгоритмы *partial_sort_copy*. Снова один из этих двух алгоритмов использует для сравнения оператор «меньше», а другой – предикат, который передается дополнительным аргументом. Первый вариант, использующий <, встречается в следующей программе:

```
// partsrt1.cpp: partial_sort_copy.
#include <iostream>
```

```
    nth_element(a, a+2, a+10);
    copy(a, a+10,  ostream_iterator<int>(cout, " "));
    cout << endl;
    return 0;
}
```

Вывод

```
4 4 5 6 6 6 9 12 8 10
```

этой программы показывает, что элемент $a[2]$ содержит значение 5. Этот элемент будет иметь такое же значение в полностью отсортированном массиве:

```
4 4 5 6 6 6 8 9 10 12
```

Если не считать порядка, содержание массива не меняется. Кроме того, все элементы, предшествующие выбранному элементу, не превышают по значению этот элемент, а все элементы, следующие за ним, не меньше его.

Три аргумента функции *nth_element* являются итераторами произвольного доступа. Вторая версия алгоритма имеет четвертый аргумент, который является предикатом для сравнения. Например, если мы заменим вызов *nth_element* в вышеприведенной программе на

```
nth_element(a, a+2, a+10, greater<int>());
```

то получим следующий вывод:

```
10 12 9 8 6 6 6 4 5 4
```

Следует обратить внимание, что $a[2] = 9$, как и в случае, если этот массив будет полностью отсортирован в нисходящем порядке. Элементы, предшествующие $a[2]$, по своему значению не меньше, а те, которые следуют после, по значению не больше этого элемента.

7.3.6. Двоичный поиск

```
ForwardIterator lower_bound
   (ForwardIterator first, ForwardIterator last,
    const T& value);
ForwardIterator lower_bound
   (ForwardIterator first, ForwardIterator last,
    const T& value, Compare comp);
ForwardIterator upper_bound
   (ForwardIterator first, ForwardIterator last,
    const T& value);
ForwardIterator upper_bound
   (ForwardIterator first, ForwardIterator last,
    const T& value, Compare comp);
```

```
pair<ForwardIterator, ForwardIterator> equal_range
   (ForwardIterator first, ForwardIterator last,
    const T& value);
pair<ForwardIterator, ForwardIterator> equal_range
   (ForwardIterator first, ForwardIterator last,
    const T& value, Compare comp);
bool binary_search
   (ForwardIterator first, ForwardIterator last,
    const T& value);
bool binary_search
   (ForwardIterator first, ForwardIterator last,
    const T& value, Compare comp);
```

В этом разделе мы обсудим четыре алгоритма – *lower_bound*, *upper_bound*, *equal_range* и *binary_search*. В нашей программе встретятся только версии, которые используют оператор сравнения <. Для каждого из этих четырех алгоритмов существует также версия, имеющая дополнительный аргумент, предикат сравнения. Все алгоритмы предполагают, что используемый диапазон отсортирован. Программа ниже использует версии четырех алгоритмов без дополнительного аргумента-предиката:

```
// bsearch.cpp: Двоичный поиск и родственные алгоритмы.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{  int a[10] = {3, 3, 5, 5, 5, 5, 5, 7, 8, 9}, *p, k;
   cout << "Array a:\n";
   for (k=0; k<10; k++) cout << k << " ";
   cout << endl;
   for (k=0; k<10; k++) cout << a[k] << " ";
   cout << endl;

   p = lower_bound(a, a+10, 5);
   cout  << "p - a = " << p - a
         << " after p = lower_bound(a, a+10, 5);\n";

   p = lower_bound(a, a+10, 4);
   cout  << "p - a = " << p - a
         << " after p = lower_bound(a, a+10, 4);\n";

   p = upper_bound(a, a+10, 5);
   cout  << "p - a = " << p - a
         << " after p = upper_bound(a, a+10, 5);\n";

   pair<int*, int*> P(0, 0);
   P = equal_range(a, a+10, 5);
   cout  <<
```

```
        "After P = equal_range(a, a+10, 5) we have:\n";
    cout  << "  P.first  - a = " << P.first  - a << endl;
    cout  << "  P.second - a = " << P.second - a << endl;

    bool b = binary_search(a, a+10, 5);
    cout  << "Results of binary_search:\n";
    cout  << "5 " << (b ? "" : "not ")
          << "found in array a.\n";
    b = binary_search(a, a+10, 4);
    cout  << "4 " << (b ? "" : "not ")
          << "found in array a.\n";
    return 0;
}
```

В результате мы получаем такой вывод:

```
Array a:
0 1 2 3 4 5 6 7 8 9
3 3 5 5 5 5 5 7 8 9
p - a = 2 after p = lower_bound(a, a+10, 5);
p - a = 2 after p = lower_bound(a, a+10, 4);
p - a = 7 after p = upper_bound(a, a+10, 5);
After P = equal_range(a, a+10, 5) we have:
   P.first  - a = 2
   P.second - a = 7
Results of binary_search:
5 found in array a.
4 not found in array a.
```

Как показывает этот вывод, алгоритм *lower_bound* возвращает итератор, ссылающийся на *первую* позицию, куда можно вставить заданный элемент без нарушения порядка сортировки. Для значений 4 и 5 это позиция $a[2]$. Аналогично вызов *upper_bound* возвращает значение, указывающее, что $a[7]$ является *последней* позицией, куда может быть поставлено значение 5 без нарушения сортировки.

Вызов *equal_range* сообщает нам, что $[2, 7)$ является диапазоном позиций, куда может быть вставлено значение 5. Если бы мы использовали вызов

```
P = equal_range(a, a+10, 4);
```

тогда значения обоих выражений *P.first* – *a* и *P.second* – *a* были бы равны 2, поскольку $a[2]$ – единственная позиция, куда может быть добавлено значение 4.

Алгоритм *binary_search* информирует нас, найдено ли искомое значение в диапазоне, но не сообщает, в какой позиции находится это значение, ни в какую позицию его можно добавить, если оно не найдено.

7.3.7. Объединение

```
OutputIterator merge
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);
OutputIterator merge
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, Compare comp);
void inplace_merge
   (BidirectionalIterator first,
    BidirectionalIterator middle,
    BidirectionalIterator last);
void inplace_merge
   (BidirectionalIterator first,
    BidirectionalIterator middle,
    BidirectionalIterator last, Compare comp);
```

Существуют две версии алгоритма *merge* (объединить), одну из которых мы обсуждали в разделе 1.7. Другая версия имеет шестой аргумент, являющийся предикатом. Она может быть использована для объединения двух диапазонов, отсортированных в порядке убывания, как показывает следующая программа:

```
// merge1.cpp:   Объединение двух диапазонов,
//               отсортированных в нисходящем порядке.
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{  int   a[5] = {30, 28, 15, 13, 11},
         b[4] = {25, 24, 15, 12},
         c[9];
   merge(a, a+5, b, b+4, c, greater<int>());
   copy(c, c+9, ostream_iterator<int>(cout, " "));
   cout << endl;
   return 0;
}
```

Эта программа выводит:

```
30 28 25 24 15 15 13 12 11
```

Напомним, что мы можем опустить последний аргумент в вызове *merge*, если массивы *a* и *b* отсортированы в восходящем порядке.

Если две отсортированные в восходящем порядке последовательности находятся в одном диапазоне [*first*, *last*), причем одна из них – [*first*, *middle*), а другая – [*middle*, *last*), мы можем использовать следующий вызов, чтобы объединить эти последовательности, так что диапазон [*first*, *last*) будет полностью отсортирован:

```
inplace_merge(first, middle, last);
```

Приведенная ниже программа демонстрирует работу алгоритма *inplace_merge*:

```
// inplace.cpp: Объединение двух
//              отсортированных поддиапазонов.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{  int a[7] =  {2, 5, 8, /* Второй диапазон: */
             3, 4, 5, 9};
   inplace_merge(a, a+3, a+7);
   copy(a, a+7,
   ostream_iterator<int>(cout, " "));
   cout << endl;
   return 0;
}
```

Программа выводит отсортированный полный диапазон:

```
2 3 4 5 5 8 9
```

Существует также версия алгоритма *inplace_merge*, принимающая в качестве дополнительного аргумента предикат, так что мы можем использовать нашу собственную операцию сравнения, как в программе *merge*1.

7.3.8. Операции над множествами для сортированных контейнеров

```
bool includes
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2);
bool includes
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    Compare comp);
OutputIterator set_union
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);
```

```
OutputIterator set_union
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, Compare comp);
OutputIterator set_intersection
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);
OutputIterator set_intersection
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, Compare comp);
OutputIterator set_difference
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);
OutputIterator set_difference
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, Compare comp);
OutputIterator set_symmetric_difference
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);
OutputIterator set_symmetric_difference
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result, Compare comp);
```

Мы использовали алгоритмы *set_intersection* и *set_union* для множеств в разделе 4.3. Однако они работают не только с множествами, но и с сортированными последовательными контейнерами. К сортированным структурам применимы следующие операции над множествами: *includes*, *set_union*, *set_intersection*, *set_difference* и *set_symmetric_difference*. Каждый из этих алгоритмов существует в двух версиях: одна использует для сравнения оператор <, а другая – передаваемый в качестве аргумента предикат. Следующая программа демонстрирует эти пять алгоритмов в варианте, основанном на операторе «меньше»:

```
// setopstr.cpp: Операции над множествами
//               для сортированных контейнеров.
#include <iostream>
#include <algorithm>
using namespace std;

void show ( const char *s, const int *begin,
         const int *end)
```

```
{  cout << s << " ";
   copy(begin, end, ostream_iterator<int>(cout, " "));
   cout << endl;
}

int main()
{  int   a[4] = {1, 5, 7, 8}, b[3] = {2, 5, 8},
         sum[7], *pSumEnd,
         prod[4], *pProdEnd,
         dif[3], *pDifEnd,
         symdif[7], *pSymDifEnd;
   pSumEnd = set_union(a, a+4, b, b+3, sum);
   pProdEnd = set_intersection(a, a+4, b, b+3, prod);
   pDifEnd = set_difference(a, a+4, b, b+3, dif);
   pSymDifEnd = set_symmetric_difference(a, a+4, b, b+3,
      symdif);
   show("a:     ", a, a+4);
   show("b:     ", b, b+3);
   show("sum:   ", sum, pSumEnd);
   show("prod:  ", prod, pProdEnd);
   show("dif:   ", dif, pDifEnd);
   show("symdif:", symdif, pSymDifEnd);
   if (includes(a, a+4, b, b+3))
      cout << "a includes b.\n";
   else cout << "a does not include b.\n";
   if (includes(sum, pSumEnd, b, b+3))
      cout << "sum includes b.\n";
   else cout << "sum does not include b.\n";
   return 0;
}
```

Эта программа выводит следующий текст:

```
a:      1 5 7 8
b:      2 5 8
sum:    1 2 5 7 8
prod:   5 8
dif:    1 7
symdif: 1 2 7
a does not include b.
sum includes  b.
```

Переменная *sum* используется для объединений множеств *a* и *b*, а переменная *prod* – для их пересечения. Любой элемент, входящий в *a* и *b*, также входит и в *sum*; в противоположность этому в *prod* вошли только те элементы, которые входят в оба контейнера. Разность *dif* множеств *a* и *b* состоит из всех элементов, которые входят в *a*, но не в *b*. Наконец симметричная разность *symdif* состоит из всех элементов, принадлежащих только к одному из множеств *a* или *b*, но не к обоим сразу.

Говорят, что множество *S* содержит (*include*) множество *T*, если все элементы *T* входят также и в *S*. В нашем примере *a* не содержит *b*, потому что элемент 2 множества *b* не принадлежит *a*. Объединение (*union*) двух множеств всегда содержит каждое из этих множеств, поэтому в нашем примере *sum* содержит *b*.

Мы можем использовать последовательные контейнеры, такие как массивы *a* и *b* из нашего примера, для представления «множеств с дубликатами» и также применять к ним алгоритмы типа *set_union* и др. Получающееся объединение будет контейнером, который содержит максимум повторений каждого элемента в любом из двух множеств[1], пересечение будет содержать минимум повторений. Давайте заменим в только что рассмотренной программе инициализацию переменной *a*, в результате чего вторая строчка функции *main* будет выглядеть вот так:

```
{  int a[4] = {1, 1, 7, 8}, b[3] = {2, 5, 8},
```

После такого изменения мы получим следующий вывод:

```
a:      1 1 7 8
b:      2 5 8
sum:    1 1 2 5 7 8
prod:   8
dif:    1 1 7
symdif: 1 1 2 5 7
a does not include b.
sum includes b.
```

7.3.9. Операции над пирамидами

```
void push_heap
   (RandomAccessIterator first,
    RandomAccessIterator last);
void push_heap
   (RandomAccessIterator first,
    RandomAccessIterator last, Compare comp);
void pop_heap
   (RandomAccessIterator first,
    RandomAccessIterator last);
void pop_heap
   (RandomAccessIterator first,
    RandomAccessIterator last, Compare comp);
```

[1] Обратите внимание на отличие поведения операции объединения множеств (*set_union*) от операции объединения сортированных контейнеров (*merge*). Для последней количество повторений элемента в результате будет равно *сумме* количеств повторений элемента в каждом из исходных контейнеров. – *Прим. переводчика.*

```
void make_heap
   (RandomAccessIterator first,
    RandomAccessIterator last);
void make_heap
   (RandomAccessIterator first,
    RandomAccessIterator last, Compare comp);
void sort_heap
   (RandomAccessIterator first,
    RandomAccessIterator last);
void sort_heap
   (RandomAccessIterator first,
    RandomAccessIterator last, Compare comp);
```

Пирамида (*heap*) – это особый способ организации элементов в диапазоне [*start*, *end*), где *start* и *end* – итераторы произвольного доступа. Рассмотрим следующий пример пирамиды:

```
i ->     9  1  2  3  4  5  6  7  8  9
a[i] -> 80 70 60 40 50 45 30 25 20 10
```

В этом примере, как и в любой пирамиде из десяти элементов, у нас:

$a[0]$ не меньше, чем $a[1]$ и $a[2]$,
$a[1]$ не меньше, чем $a[3]$ и $a[4]$,
$a[2]$ не меньше, чем $a[5]$ и $a[6]$,
$a[3]$ не меньше, чем $a[7]$ и $a[8]$,
$a[4]$ не меньше, чем $a[9]$.

Вообще говорят, что контейнер a с элементами $a[0]$, $a[1]$, ..., $a[n - 1]$ удовлетворяет *условию пирамидальности*, если

$$a[i] \geqslant a[2 * i + 1]$$
$$a[i] \geqslant a[2 * i + 2]$$

для всех элементов, принадлежащих контейнеру. Отсюда следует, что первый элемент ($a[0]$) пирамиды является наибольшим. Пирамиды удобно использовать для реализации очередей с приоритетами, рассмотренных в разделе 5.3, поскольку существуют операции эффективного извлечения первого элемента и добавления нового элемента с сохранением условия пирамидальности. Извлечение первого элемента пирамиды в STL реализовано следующим образом: сначала мы копируем первый элемент как обычно, а затем вызываем алгоритм *pop_heap*. Например, для массива *a* мы можем написать

```
x = *a; pop_heap(a, a+10);
```

а для вектора или двусторонней очереди *v*:

```
x = *v.begin(); pop_heap(v.begin(), v.end());
```

Функция *pop_heap* логически удаляет первый элемент контейнера, а затем восстанавливает условие пирамидальности.

Добавляем новый элемент в пирамиду мы также с помощью двух операций: сперва помещаем новый элемент в конце пирамиды, а затем вызываем алгоритм *push_heap*. Например:

```
a[9] = x; push_heap(a, a+10);
```

или

```
v.push_back(x); push_heap(v.begin(), v.end());
```

Алгоритм *push_heap* восстанавливает условие пирамидальности, если только последний элемент последовательности нарушает это условие. Существует более мощная (но требующая больше времени для выполнения) функция *make_heap*, которая превращает последовательный контейнер с произвольным доступом в пирамиду. И наконец, мы можем использовать алгоритм *sort_heap* для превращения пирамиды в отсортированную последовательность. Обратите внимание, что этот алгоритм помещает элементы в восходящем порядке, так что они перестают удовлетворять условию пирамидальности. Упомянутые выше алгоритмы работы с пирамидами используются в следующей программе:

```
// heapdemo.cpp: Демонстрация операций с пирамидами.
#include <iostream>
#include <algorithm>
using namespace std;

void show(  const char *s, const int *begin,
         const int *end)
{  cout << s << endl << "   ";
   copy(begin, end,  ostream_iterator<int>(cout, " "));
   cout << endl;
}

int main()
{  cout << "  ";
   for (int i=0; i<10; i++) cout << "  " << i;
   cout << endl;
   int a[10] = {20, 50, 40, 60, 80, 10, 30, 70, 25, 45};
   show("Initial contents of a:", a, a+10);
   random_shuffle(a, a+10);
   show("After random_shuffle(a, a+10):", a, a+10);
   make_heap(a, a+10);
```

```
    show("After make_heap(a, a+10):", a, a+10);

    int x = *a;
    pop_heap(a, a+10);
    show("After x = *a and pop_heap(a, a+10):", a, a+9);
    a[9] = x;
    push_heap(a, a+10);
    show("After a[9] = x and push_heap(a, a+10):",
       a, a+10);
    sort_heap(a, a+10);
    show("After sort_heap(a, a+10):", a, a+10);
    return 0;
}
```

Эта программа выводит на экран следующие результаты, которые являются иллюстрацией к нашему обсуждению:

```
    0  1  2  3  4  5  6  7  8  9
Initial contents of a:
   20 50 40 60 80 10 30 70 25 45
After random_shuffle(a, a+10):
   80 10 45 25 50 60 30 20 40 70
After make_heap(a, a+10):
   80 70 60 40 50 45 30 20 25 10
After x = *a and pop_heap(a, a+10):
   70 50 60 40 10 45 30 20 25
After a[9] = x and push_heap(a, a+10):
   80 70 60 40 50 45 30 20 25 10
After sort_heap(a, a+10):
   10 20 25 30 40 45 50 60 70 80
```

После вызова *make_heap* первый элемент наибольший, но последовательность не является отсортированной в нисходящем порядке; например, 40 предшествует 50. Однако она является пирамидой, поэтому, после копирования первого элемента (80) в переменную *x*, мы можем использовать алгоритм *pop_heap*, чтобы получить пирамиду из девяти элементов. Это значение (80) снова добавляется в пирамиду: сначала мы помещаем его в конец последовательности, а затем вызываем алгоритм *push_heap* для восстановления условия пирамидальности. Кроме всего, пирамида сортируется с помощью специального алгоритма *sort_heap*, который использует свойства пирамиды и поэтому работает быстрее, чем алгоритм общего назначения *sort*.

7.3.10. Минимум и максимум

```
const T& min
   (const T& a, const T& b);
```

```
const T& min
   (const T& a, const T& b, Compare comp);
const T& max
   (const T& a, const T& b);
const T& max
   (const T& a, const T& b, Compare comp);
ForwardIterator min_element
   (ForwardIterator first, ForwardIterator last);
ForwardIterator min_element
   (ForwardIterator first, ForwardIterator last,
    Compare comp);
ForwardIterator max_element
   (ForwardIterator first, ForwardIterator last);
ForwardIterator max_element
   (ForwardIterator first, ForwardIterator last,
    Compare comp);
```

Если мы хотим найти минимум или максимум двух объектов, для которых определен оператор <, то можем использовать алгоритмы *min* и *max*, реализовать которые можно, например, следующим хорошо известным способом:

```
#define min(x, y) ((x) < (y) ? (x) : (y))
#define max(x, y) ((x) < (y) ? (y) : (x))
```

Существуют также версии алгоритмов *min* и *max* с третьим аргументом, задающим операцию сравнения. Следующая программа использует *min* с двумя и *max* с тремя элементами:

```
// minmax.cpp: Алгоритмы min и max.
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

bool CompareLastDigit(int x, int y)
{  return x % 10 < y % 10;
}

int main()
{  int x = 123, y = 75, minimum, MaxLastDigit;
   minimum = min(x, y);
   MaxLastDigit = max(x, y, CompareLastDigit);
   cout << minimum << " " << MaxLastDigit << endl;
   return 0;   // Вывод: 75 75
}
```

В этой программе вызов *min* возвращает 75, меньшее из чисел 123 и 75. Вызов *max* также возвращает 75, потому что последняя цифра (5) этого числа больше, чем последняя цифра числа 123.

Чтобы найти позицию наименьшего элемента в последовательности, мы можем использовать алгоритм *min_element*. В следующей программе этот алгоритм используется для массива и для списка:

```
// min_elt.cpp: Наименьший элемент последовательности.
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{  int a[5] = {10, 30, 5, 40, 20}, *p;
   list<int> L;
   L.insert(L.begin(), a, a+5);
   list<int>::iterator i;
   p = min_element(a, a+5);
   i = min_element(L.begin(), L.end());
   cout << *p << " " << *i << endl;
   // Вывод: 5 5
   return 0;
}
```

Существует также версия алгоритма *min_element,* которая имеет третий аргумент для передачи ему операции сравнения аналогично тому, как это сделано для алгоритмов *min* и *max*. Если мы хотим найти позицию максимального элемента в последовательности, можно использовать алгоритм *max_element*, который имеет те же два варианта, что и *min_element*.

7.3.11. Лексикографическое сравнение

```
bool lexicographical_compare
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2);
bool lexicographical_compare
   (InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    Compare comp);
```

Мы можем проводить лексикографическое сравнение последовательностей аналогично тому, как мы сравниваем строки текста: если первые элементы различны, они определяют результат сравнения, в противном случае мы сравниваем вторые элементы и т. д. Существуют два алгоритма *lexicographical_compare*: один с четырьмя параметрами использует для сравнения элементов операцию <, другой же принимает операцию сравнения в качестве дополнительного аргумента. Пример работы этих алгоритмов приведен в следующей программе:

```
// lexcomp.cpp: Лексикографическое сравнение.
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{  int   a[4] = {1, 3, 8, 2},
      b[4] = {1, 3, 9, 1};

   cout << "a: ";
   copy(a, a+4, ostream_iterator<int>(cout, " "));

   cout << "\nb: ";
   copy(b, b+4, ostream_iterator<int>(cout, " "));
   cout << endl;

   if (lexicographical_compare(a, a+4, b, b+4))
      cout << "Lexicographically, a precedes b.\n";

   if (lexicographical_compare(b, b+4, a, a+4,
      greater<int>()))
      cout <<
      "Using the greater-than relation, we find:\n"
      "b lexicographically precedes a.\n";
   return 0;
}
```

Результат работы программы показан ниже:

```
a: 1 3 8 2
b: 1 3 9 1
Lexicographically, a precedes b.
Using the greater-than relation, we find:
b lexicographically precedes a.
```

Если последовательности имеют разную длину, отсутствующий в последовательности элемент предшествует любому присутствующему элементу второй последовательности; например: {1, 2} лексикографически предшествует {1, 2, –5}, вне зависимости от оператора сравнения.

7.3.12. Генераторы перестановок

```
bool next_permutation
   (BidirectionalIterator first,
    BidirectionalIterator last);
bool next_permutation
   (BidirectionalIterator first,
    BidirectionalIterator last,
    Compare comp);
```

```
bool prev_permutation
   (BidirectionalIterator first,
    BidirectionalIterator last);
bool prev_permutation
   (BidirectionalIterator first,
    BidirectionalIterator last,
    Compare comp);
```

Алгоритм *next_permutation* порождает очередную перестановку последовательности. Последовательность вызовов этого алгоритма порождает, если это возможно, перестановки в лексикографическом порядке. Возвращаемое значение типа *bool* показывает, существует ли следующая перестановка. Это поведение алгоритма становится ясным из следующей программы; она также показывает, как алгоритм *prev_permutation* порождает предшествующую перестановку:

```
// permgen.cpp: Генератор перестановок порождает все
//              перестановки последовательности 1 2 3.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{  int a[3] = {1, 2, 3}, k;
   cout << "Six successive calls to next_permutation.\n"
      "Situation before call and value returned by "
      "call:\n";
   for (k=0; k<6; k++)
   {  copy(a, a+3, ostream_iterator<int>(cout, " "));
      bool b = next_permutation(a, a+3);
      cout << (b ? " true" : " false") << endl;
   };

   cout <<
      "Three successive calls to prev_permutation.\n"
      "Situation before call and value returned by "
      "call:\n";
   for (k=0; k<3; k++)
   {  copy(a, a+3, ostream_iterator<int>(cout, " "));
      bool b = prev_permutation(a, a+3);
      cout << (b ? " true" : " false") << endl;
   };
   return 0;
}
```

Начиная с последовательности {1, 2, 3}, программа выводит содержимое массива *a* непосредственно *перед* вызовом функции *next_permutation*. Всего

проходит шесть таких вызовов. Первый вызов возвращает *true* и изменяет последовательность, содержащуюся в *a*, на {1, 3, 2}, второй вызов также возвращает *true* и порождает последовательность {2, 1, 3} и т. д. Как видно из приведенного ниже вывода, перестановки появляются в лексикографическом порядке. Когда *a* = {3, 2, 1}, шестой вызов функции *next_permutation* возвращает *false* и помещает в массив исходную последовательность {1, 2, 3}. Функция *prev_permutation* работает в обратном порядке. Она возвращает *false*, когда изменяет содержимое массива с {1, 2, 3} на {3, 2, 1}, и *true* при переходе от {3, 2, 1} к {3, 1, 2}, от {3, 1, 2} к {2, 3, 1} и т. д.:

```
Six successive calls to next_permutation.
Situation before call and value returned by call:
1 2 3  true
1 3 2  true
2 1 3  true
2 3 1  true
3 1 2  true
3 2 1  false
Three successive calls to prev_permutation.
Situation before call and value returned by call:
1 2 3  false
3 2 1  true
3 1 2  true
```

Кроме рассмотренных нами существуют версии функций *next_permutation* и *prev_permutation*, которые имеют дополнительный параметр для задания операции сравнения.

7.4. Обобщенные численные алгоритмы

Чтобы использовать алгоритмы, обсуждаемые в этом разделе, мы должны написать в программе

```
#include <numeric>
```

если работаем с версией STL, соответствующей проекту стандарта C++. Для HP STL необходимо заменить эту строчку на следующую:

```
#include <algo.h>
```

7.4.1. Накопление

```
T accumulate
   (InputIterator first, InputIterator last, T init);
T accumulate
   (InputIterator first, InputIterator last, T init,
    BinaryFunction binary_op);
```

грамме эта версия алгоритма используется для вычисления произведения степеней.

```
// inprod2.cpp: Произведение степеней, вычисляемое
//              с помощью алгоритма inner_product.

#include <iostream>
#include <numeric>
#include <functional>
using namespace std;

double power(int x, int n)
{  double y = 1;
   for (int k=0; k<n; k++) y *= x;
   return y;  // x в степени n
}

int main()
{  int a[3] = {2, 3, 5}, b[3] = {4, 1, 2}, product=1;
   product = inner_product(a, a+3, b, product,
      multiplies<double>(), power);
   cout << product << endl;
   // 1 * power(2, 4) * power(3, 1) * power(5, 2) =
   // 1 * 16 * 3 * 25 = 1200
   return 0;
}
```

7.4.3. Частичная сумма

```
OutputIterator partial_sum
   (InputIterator first, InputIterator last,
    OutputIterator result);
OutputIterator partial_sum
   (InputIterator first, InputIterator last,
    OutputIterator result, BinaryOperation binary_op);
```

Алгоритм *partial_sum* практически рассчитывает суммы элементов с накоплением. Например, из последовательности {2, 3, 4, 8} мы получим {2, 5, 9, 17}, копируя первое значение, 2, и вычисляя три других элемента следующим образом:

2 + 3 = 5
5 + 4 = 9
9 + 8 = 17

Приведенная ниже программа использует алгоритм *partial_sum* для вычисления значений в рассмотренном примере:

```
// partsum.cpp: Частичная сумма.
#include <iostream>
```

```
#include <numeric>
#include <algorithm>
using namespace std;

int main()
{  int a[4] = {2, 3, 4, 8}, b[4], *iEnd;
   iEnd = partial_sum(a, a+4, b);
   copy(b, iEnd, ostream_iterator<int>(cout, " "));
   cout << endl; // Вывод 2 5 9 17
   return 0;
}
```

Алгоритм *partial_sum* возвращает итератор, который ссылается на элемент, следующий за концом результирующей последовательности. Другими словами, *iEnd* равно значению $b + 4$ в этой программе. Результирующие суммы могут быть помещены в исходном контейнере, поэтому нам не потребуются два различных массива a и b, если мы хотим получить результат в массиве a; для этого достаточно написать

```
iEnd = partial_sum(a, a+4, a);
```

Существует другая версия *partial_sum*, которая принимает дополнительный параметр, позволяющий заменить оператор + на другую операцию. Например, мы можем написать

```
iEnd = partial_sum(a, a+4, b, multiplies<int>());
```

вместо вызова *partial_sum* в приведенной выше программе, чтобы получить в результате 2 6 24 192, поскольку

$$2 \times 3 = 6$$
$$6 \times 4 = 24$$
$$24 \times 8 = 192$$

7.4.4. Разность между смежными элементами

```
OutputIterator adjacent_difference
   (InputIterator first, InputIterator last,
    OutputIterator result);
OutputIterator adjacent_difference
   (InputIterator first, InputIterator last,
    OutputIterator result,
    BinaryFunction binary_op);
```

Алгоритм *adjacent_difference* (смежная разность) вычисляет разности

$$d_i = a_i - a_{i-1}$$

для $i > 0$ и $d_0 = a_0$. Например, для массива $a = \{2, 3, 4, 8\}$ мы получим разности $d = \{2, 1, 1, 4\}$, потому что $3 - 2 = 1$, $4 - 3 = 1$ и $8 - 4 = 4$. Алгоритм

возвращает значение итератора «за концом результата», что демонстрируется в следующей программе:

```
// adjdif.cpp: Разность между смежными элементами.

#include <iostream>
#include <numeric>
#include <algorithm>
using namespace std;

int main()
{  int a[4] = {2, 3, 4, 8}, d[4], *iEnd;
   iEnd = adjacent_difference(a, a+4, d);
   copy(d, iEnd, ostream_iterator<int>(cout, " "));
   cout << endl; // Вывод 2 1 1 4
   return 0;
}
```

Чтобы результат был сохранен в исходной последовательности, мы можем написать

```
iEnd = adjacent_difference(a, a+4, a);
```

Существует другая версия алгоритма, принимающая дополнительный параметр, чтобы можно было указать операцию, отличную от вычитания. Например, после выполнения

```
int a[4] = {2, 20, 150, 700}, b[4], *iEnd;
iEnd = adjacent_difference(a, a+4, b, divides<int>());
```

мы получим $b = \{2, 15, 5, 4\}$, так как $30/2 = 15$, $150/30 = 5$ и $700/150 = 4$.

7.5. Прикладная программа: метод наименьших квадратов

Теперь мы применим алгоритмы *accumulate* и *inner_product* для решения известной практической задачи. Предположим, что имеется набор из n пар чисел (x_i, y_i), где каждая пара соответствует точке на плоскости xy, и мы хотим найти прямую линию, которая является достаточно разумным приближением зависимости, представленной этими точками, как показано на рисунке 7.1.

Количество точек произвольно, но не может быть меньше 2. Расположение точек должно быть таким, чтобы получающаяся линия не оказалась вертикальной; в частности недопустимо, если все точки имеют одну и ту же координату x.

Координаты n точек будут находиться во входном файле, имя которого вводится пользователем. Например, восемь точек, изображенных в виде крестиков на рисунке 7.1, находятся в следующем файле *data.txt*:

- алгоритм *accumulate* позволит просто вычислить значения $\sum x_i$ и $\sum y_i$;
- алгоритм *inner_product* позволит просто вычислить $\sum x_i^2$ и $\sum x_i y_i$;
- последовательные контейнеры позволяют хранить переменное количество значений x и y.

Что касается последнего пункта: на самом деле нет необходимости хранить все эти числа, обычно это и не делалось в раннюю эпоху компьютерных вычислений, когда память компьютеров была ограничена. Мы предположим, что располагаем достаточным количеством памяти для хранения всех чисел; более того, это позволит нам выводить не только искомые коэффициенты a и b, но и список отклонений

$$y_i - y(i)$$

где, как уже упоминалось, $y(i) = a + bx_i$. Если бы мы не сохраняли все значения, нам пришлось бы дважды читать входной файл. Используя тип *vector<double>* для хранения значений x и y, получаем следующую программу:

```
// leastsq.cpp: Метод наименьших квадратов.
#include <iostream>
#include <fstream>
#include <iomanip>
#include <vector>
#include <numeric>
#include <string>
using namespace std;
typedef vector<double> array;

int ReadPoints(ifstream &file, array &x, array &y)
{  double x1, y1;
   while (file >> x1 >> y1)
   {  x.push_back(x1); y.push_back(y1);
   }
   return x.size();
}

int ComputeCoeff(const array &x, const array &y,
   double &a, double &b)
{  double sx=0, sx2=0, sy=0, sxy=0;
   sx = accumulate(x.begin(), x.end(), sx);
   sy = accumulate(y.begin(), y.end(), sy);
   sx2 = inner_product(x.begin(), x.end(),
      x.begin(), sx2);
   sxy = inner_product(x.begin(), x.end(),
      y.begin(), sxy);
```

```
   int n = x.size();
   double D = n * sx2 - sx * sx;
   if (D != 0)
   {  a = (sy * sx2 - sx * sxy)/D;
      b = (n * sxy - sx * sy)/D;
      return 1;
   }  else return 0;
}

void ShowComputedPoints(const array &x, const array &y,
   double a, double b)
{  cout << "\n         x     yGiven  yComputed\n\n"
      << setiosflags(ios::fixed);
   int n = x.size();
   for (int i=0; i<n; i++)
      cout << setw(10) << setprecision(2)<< x[i]
         << " " << setw(10) << y[i]
         << " " << setw(10) << a + b * x[i] << endl;
}

int main()
{  array x, y;
   double a, b;
   string FileName;
   cout << "Input file: ";
   cin >> FileName;
   ifstream file(FileName.c_str());
   if (!file)
   {  cout << "Cannot open input file.\n";
      return 1;
   }
   int n = ReadPoints(file, x, y);
   cout << "n = " << n << endl;
   if (n < 2)
   {  cout << "Too few points.\n";
      return 1;
   }
   if (ComputeCoeff(x, y, a, b) == 0)
   {  cout << "Vertical line not allowed.\n";
      return 1;
   }
   cout << "Regression line is y = a + bx, where\n"
      << "a = " << a << " and b = " << b << endl;
   ShowComputedPoints(x, y, a, b);
   return 0;
}
```

Для файла *data.txt*, содержание которого приведено раньше, мы получаем показанные ниже результаты. Первые два столбца содержат заданные точки (*x*, *yGiven*), в то время как первый и третий столбцы представляют вычисленные точки (*x*, *yComputed*), которые принадлежат линии регрессии, изображенной на рисунке 7.1:

```
Input file: data.txt
n = 8
Regression line is y = a + bx, where
a = 100.8 and b = 0.845528

         x     yGiven  yComputed

     20.50     118.10     118.13
     20.60     118.25     118.22
     20.65     118.20     118.26
     20.75     118.40     118.34
     20.80     118.40     118.39
     20.85     118.50     118.43
     20.90     118.45     118.47
     21.00     118.50     118.56
```

Типы начальных значений

Обратим внимание на то, что начальные значения, например третий аргумент при вызове

```
sx = accumulate(x.begin(), x.end(), sx);
```

задаются в программе с помощью инициализированных переменных, хотя мы могли бы написать вместо этого

```
sx = accumulate(x.begin(), x.end(), 0.0);
```

и не инициализировать переменную *sx* нулем. Последний вызов алгоритма *accumulate* является правильным, потому что константа 0.0 принадлежит к типу *double*. Но стоит отметить, что программа дает неверный результат, если в этом вызове мы заменим 0.0 на значение 0 типа *int*. Простой способ избежать таких проблем с неверным типом константы – использовать для начального значения ту же переменную, в которой сохраняется результат работы алгоритма. Этот принцип относится как к алгоритму *accumulate*, так и к алгоритму *inner_product*.

8

Прикладная программа: очень большие числа

8.1. Введение

Эта глава сильно отличается от предыдущих. В ней мы рассмотрим полезный на практике класс *large*, реализующий операции с очень большими числами, для чего он использует контейнеры и алгоритмы STL. Кроме достаточно сложного файла определения, *large.cpp*, для этого класса имеется соответствующий заголовок – *large.h*. Класс *large* используется двумя прикладными программами. Программа *largedem.cpp* демонстрирует все доступные операции. После этого мы рассмотрим более интересное приложение, программу *largepi.cpp*, которая вычисляет математическую константу π с любым заданным количеством знаков после запятой.

Тема настоящей главы была рассмотрена в предыдущей книге *Algorithms and Data Structures in C++* (см. библиографию), но без использования STL. Приведенный здесь вариант решения получился проще и элегантнее благодаря использованию вектора STL. Как мы упоминали в разделе 3.7, такая реализация позволяет обойтись без определения конструктора копирования и оператора присваивания. В текстах программ этой главы не используются операторы *new* и *delete*, тогда как версия из предыдущей

книги использовала оператор *new* восемь раз в файле *large.cpp* и один раз в *largepi.cpp*.

Представление чисел

Наши целые числа неограниченного размера будут представлены в виде двоичных чисел. Их удобно представлять записанными в системе счисления с основанием $B = 2^n$, где n – длина машинного слова. Мы будем группировать вместе последовательности n бит. Это напоминает обычный прием группировки четырех бит для записи битовых последовательностей в виде шестнадцатеричных чисел. Количество бит, используемое для представления числа в классе *large*, всегда будет кратным n. Для 32-битных целых чисел (то есть когда $n = 32$ и *sizeof*(*int*) = 4) «цифры» будут находиться в диапазоне от 0 до $2^{32} - 1 = 4\ 294\ 967\ 295$.

Если мы увеличим последнее значение на 1, то не сможем хранить результат в одной «цифре» – нам потребуются уже две цифры или 64 бита. Каждое число у нас будет представлено вектором, содержащим значения типа *unsigned int* вместе с флагом *neg* типа *bool*, который равен *true*, если данное число отрицательно, и *false*, если оно положительно или равно 0. Каждый элемент вектора будет содержать одну «цифру» в диапазоне от 0 до 2^{n-1}. Для 32-битных целых чисел размер вектора в зависимости от значения представленного числа x будет следующим:

0, если $x = 0$,
1, если $0 < |x| < 2^{32}$,
2, если $2^{32} \leqslant |x| < 2^{64}$,
3, если $2^{64} \leqslant |x| < 2^{96}$,
4, если $2^{96} \leqslant |x| < 2^{128}$.

и так далее. Класс *large* будет определять много операций, большая часть которых доступна и для обычных арифметических типов:

- арифметические операции +, –, *, /, % (и функцию *divide*, объединяющую / и %);
- операции битового сдвига << и >>;
- все перечисленные выше операции с одновременным присваиванием, что дает +=, –=, *=, /=, %=, <<= и >>=;
- обычные перегруженные операторы ввода и вывода << и >>;
- присваивание =;
- сравнение ==, !=, <, >, <=, >=;
- унарная операция изменения знака –;
- функции *abs* для получения абсолютной величины числа, *sqrt* для вычисления квадратного корня и *power* для возведения в степень;

- преобразование каждого из типов *int*, *unsigned int*, *long* и *unsigned long* к типу *large*. Кроме этого, определяется преобразование в тип *large* обычных строк в стиле C, *char** (если эти строки содержат только десятичные цифры, которым может предшествовать знак минус);
- функция *num2char* для преобразования числа типа *large* в символьное представление, записанное в экземпляре класса *vector<char>* в обратном порядке (см. замечание о переносимости в конце этого раздела).

Большая часть этих операций используется в демонстрационной программе *largedem.cpp*, которая приведена ниже.

```
// largedem.cpp: Используем тип large.
#include "large.h"
int main()
{  large a = -10000, b = 10000U, c = 2000000L,
      d = "100000000000000000000", // 20 нулей
      x, y, z, u;
   x = (a * b * b + 1) * c;
   x -= c;       // x = a * b * b * c
   x /= a * b;   // x = b * c
   y = large("1234567890123") % large("1234567890000");
   if (x == b * c && y == large(123))
      cout << "Arithmetic OK" << endl;

   z = power(d, 100);  // d в степени 100
                       // = 10 в степени 2000
   u = sqrt(z);        // 10 в степени 1000
   if (u == power(large(10), 1000))
      cout  << "u = '10 raised to the power 1000'"
            << endl;

   if    (u < power(large(11), 1000) &&
       u > power(large(9), 1000))
      cout << "Comparisons OK" << endl;

   vector<char> s;
   u.num2char(s);
   cout  << "First character in output of u: "
         << *(s.end() - 1) << endl;
   cout  << "u consists of " << s.size()
         << " decimal digits." << endl;

   z = d << 100; // z = d * (2 в степени 100)
   if (z == d * power(large(2), 100) && (z >> 100) == d)
      cout << "Shift operations OK" << endl;
      cout << -d << endl;

   cout << "Enter a large number x: ";
   cin  >> x;
```

```
    cout << "2 * x = " << 2 * x << endl;
    a = "123456789123456789"; b = "999888777666";
    large q, r;
    a.divide(b, q, r, true);
    if (q != a/b || r != a%b)
       cout << "Function 'divide' incorrect." << endl;
    return 0;
}
```

Вот вывод этой программы:

```
Arithmetic OK
u = '10 raised to the power 1000'
Comparisons OK
First character in output of u: 1
u consists of 1001 decimal digits.
Shift operations OK
-100000000000000000000
Enter a large number x: 99988877766655544433322211
2 * x = 199977755533311088866644422
```

Рассмотренная программа показывает, что класс *large* упрощает работу с большими числами. Например, целое число $d = 10^{20}$ используется для вычисления

$$z = d^{100} = 10^{2000}$$
$$u = \sqrt{z} = 10^{1000}$$

После проверки следующих условий выводится сообщение *Comparisons OK*:

$$u = 10^{1000}$$
$$u < 11^{1000}$$
$$u > 9^{1000}$$

Для демонстрации операций сдвига число $d = 10^{20}$ сдвигается на 100 двоичных позиций влево, что дает

$$z = 10^{20} \times 2^{100}$$

Этот результат проверяется с помощью функции *power* для вычисления значения 2^{100} и сдвигом z на 100 позиций вправо, чтобы проверить, что результат снова оказывается равен d.

Переносимость

Программы в этом разделе были успешно протестированы со следующими версиями STL, упоминавшимися в разделе 1.2:

(1) Visual C++ (версия 5.0).
(2) Адаптация SGI STL для Visual C++ 5.0 от Учида.
(3) Borland C++ (версия 5.2).

Поскольку алгоритмы *min* и *max* вызывали проблемы с версией, помеченной (1), мы добавляем следующую строчку, чтобы проверить, используется ли эта версия:

```
#if (defined(_MSC_VER) && !defined(_SGI_MSVC))
```

8.2. Реализация класса *large*

Только что упомянутая строка препроцессора находится в следующем файле заголовка *large.h*, который мы использовали в программе *largedem.cpp*:

```
// large.h: Многоразрядная целочисленная арифметика.
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

typedef unsigned int uint;
typedef vector<uint> vec;

class large {
public:
   large(const char *str);
   large(int i);
   large(uint i=0);
   large(long i);
   large operator-()const;
   large &operator+=(const large &y);
   large &operator-=(const large &y);
   large &operator*=(int y);
   large &operator*=(uint y);
   large &operator*=(large y);
   large &operator/=(const large &divisor);
   large &operator%=(const large &divisor);
   large &operator<<=(uint k);
   large &operator>>=(uint k);
   void divide(large denom,
      large &quot, large &rem, bool RemDesired)const;

   // Функция num2char преобразует объект x класса large
   // в его символьное представление s в обратном порядке:
   void num2char(vector<char> &s) const;
   int compare(const large &y)const;
private:
```

```
    vec P;
    bool neg;
    void SetLen(int n);
    void reduce();
    void DDproduct(uint A, uint B, uint &Hi,
       uint &Lo)const;
    uint DDquotient(uint A, uint B, uint d)const;
    void subtractmul(uint *a, uint *b, int n,
       uint &q)const;

    bool normalize(large &denom, large &num,
       int &x)const;
    void unnormalize(large &rem, int x,
       bool SecondDone)const;
};

large operator+(large x, const large &y);
large operator-(large x, const large &y);
large operator*(large x, const large &y);
large operator/(large x, const large &y);
large operator%(large x, const large &y);
large operator<<(large u, uint k);
large operator>>(large u, uint k);
ostream &operator<<(ostream &os, const large &x);
istream &operator>>(istream &os, large &x);

bool operator==(const large &x, const large &y);
bool operator<(const large &x, const large &y);
bool operator!=(const large &x, const large &y);
bool operator>(const large &x, const large &y);
large abs(large x);
large sqrt(const large &a);
large power(large x, uint n);

#if (defined(_MSC_VER) && !defined(_SGI_MSVC))
template <class T>
inline const T& min(const T& a, const T& b)
{  return a < b ? a : b;
}

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

Значение большинства открытых (public) членов этого класса должно быть понятно из нашего обсуждения программы *largedem.cpp*. Обратите внимание, что мы определяем только два оператора сравнения – == и <.

Четыре остальных оператора (!=, >, <=, >=) определяются шаблонами STL, как объяснено в разделе 2.8.

В классе *large* присутствуют всего две переменные-члена – *P* и *neg*, определенные после ключевого слова *private* как

```
vec P;
bool neg;
```

Взглянув на определение *typedef* выше в файле, мы увидим, что тип *vec* означает *vector<unsigned int>*. Мы уже упоминали, что *P* представляет абсолютное значение большого числа, тогда как *neg* показывает, является ли число отрицательным.

Ранняя версия класса *large*, не использующая STL, содержала указатель типа *uint** вместо *vec P*, и в ней имелись еще две переменных-члена класса, *len* и *Len*, которые указывали логическую и физическую длину. Они соответствуют функциям-членам STL *size* и *capacity* (см. раздел 3.2). Теперь эти две длины полностью обрабатываются STL.

Мы не будем обсуждать все подробности реализации класса *large*; в отдельных местах текста присутствуют поясняющие комментарии, помогающие разобраться, как устроена эта реализация. После компиляции следующий файл должен быть скомпонован (*link*) с программой, которая использует класс *large*:

```
// large.cpp: Многоразрядная целочисленная арифметика,
//            значения 10, 20, 30 и 40 использующая STL.
#include <iostream>
#include <strstream>
#include <iomanip>
#include <stdlib.h>
#include <limits.h>
#include <string>
#include <ctype.h>
#include "large.h"

const uint uintmax = UINT_MAX;
const int wLen = sizeof(uint) * 8;  // Количество бит
const int hLen = wLen/2;
const uint rMask = (1 << hLen) - 1;
const uint lMask = uintmax - rMask;
const uint lBit = uintmax - (uintmax >> 1);

// Добавить нули или удалить элементы в конце,
// то есть в позициях старших разрядов.
void large::SetLen(int LenNew)
{  int LenOld = P.size();
   if (LenNew > LenOld)
```

```
   uint d;
   for (i=0; i<L; i++)
   {  if (i >= Ly && borrow == 0) break;
      d = P[i] - borrow;
      borrow = d > P[i];
      if (i < Ly)
      {  P[i] = d - y.P[i];
         if (P[i] > d) borrow = 1;
      }  else P[i] = d;
   }
   reduce();
   return *this;
}

large &large::operator*=(int y)
{  bool Neg = y < 0;
   if (Neg) y = -y;
   *this *= uint(y);
   if (Neg)
      neg = !neg;
   return *this;
}

large &large::operator*=(uint y)
{  int len0 = P.size(), i;
   uint Hi, Lo, dig = P[0], nextdig = 0;
   SetLen(len0 + 1);
   for (i=0; i<len0; i++)
   {  // Вычислим произведение двух цифр dig * y;
      // результатом является (Hi, Lo):
      DDproduct(dig, y, Hi, Lo);
      P[i] = Lo + nextdig;
      dig = P[i+1];
      nextdig = Hi + (P[i] < Lo);
   }
   P[i] = nextdig;
   reduce();
   return *this;
}

large &large::operator*=(large y)
{  int L = P.size(), Ly = y.P.size();
   if (L == 0 || Ly == 0) return *this = 0;
   bool DifSigns = neg != y.neg;
   if (L + Ly == 2)
   // L = Ly = 1: Произведение длиной в одну или две цифры:
   {  uint a = P[0], b = y.P[0];
      P[0] = a * b;  // Предположим произведение длиной
                     // в одну цифру:
```

```
      if (P[0] / a != b)
      {  P.push_back(0);
         DDproduct(a, b, P[1], P[0]);
         reduce();
      }
      neg = DifSigns;
      return *this;
   }
   if (L == 1)  //  && Ly > 1
   {  uint digit = P[0];
      *this = y;       // Поменять местами операнды
      *this *= digit;  // и вызвать operator*=(uint y)
   }  else
   if (Ly == 1) //  && L > 1
   {  *this *= y.P[0]; // Вызвать operator*=(uint y)
   }  else

   // Длины обоих операндов L и Ly больше 1:
   {  int lenProd = L + Ly, i, jA, jB;
      uint sumHi = 0, sumLo, Hi, Lo,
      sumLoOld, sumHiOld, carry=0;
      large x = *this;
      SetLen(lenProd); // Установить длину *this
                       // равной lenProd
      for (i=0; i<lenProd; i++)
      {  sumLo = sumHi; sumHi = carry; carry = 0;
         int max_jA = min(i, L-1);
         // jA <= i гарантирует jB >= 0
         // jA < L, поскольку в *this имеется лишь L цифр
         for (jA=max(0, i+1-Ly); jA<=max_jA; jA++)
         // jA > i - Ly гарантирует jB < Ly
         {  jB = i - jA;
            // Другое произведение цифр, влияющее на
            // позицию i (= jA + jB) произведения
            // всего числа:
            DDproduct(x.P[jA], y.P[jB], Hi, Lo);
            sumLoOld = sumLo; sumHiOld = sumHi;
            sumLo += Lo;
            if (sumLo < sumLoOld)
               sumHi++;
            sumHi += Hi;
            carry += (sumHi < sumHiOld);
         }
         P[i] = sumLo;
      }
   }
   reduce();
   neg = DifSigns;
```

```
   return *this;
}

large &large::operator/=(const large &divisor)
{  large r;
   // Разделить *this на делитель, сохраняя результат
   // в *this; 0 означает, что остаток в r
   // не обязан быть правильным:
   divide(divisor, *this, r, 0);
   return *this;
}

large &large::operator%=(const large &divisor)
{  large q;
   // Разделить *this на делитель, сохраняя результат
   // в *this; 1 означает, что остаток в r
   // должен быть правильным:
   divide(divisor, q, *this, 1);
   return *this;
}

large &large::operator<<=(uint k)
{  int q = k / wLen; // Количество полных слов.
   if (q)
   {  int i;         // Увеличить длину на q:
      SetLen(P.size() + q);
      // Сдвинуть *this на q слов влево:
      for (i=P.size()-1; i>=0; i--)
         P[i] = (i < q ? 0 : P[i - q]);
      k %= wLen;     // Теперь k содержит оставшиеся
   }                 // позиции сдвига.

   if (k)  // 0 < k < wLen:
   {  int k1 = wLen - k;
      uint mask = (1 << k) - 1;
      // маска: 00..011..1 (k единичных бит)
      SetLen(P.size() + 1);
      // Каждый из P[i] сдвигается на k позиций влево,
      // а затем комбинируется по "или" с k самыми левыми
      // битами правого соседа P[i-1]:
      for (int i=P.size()-1; i>=0; i--)
      {  P[i] <<= k;
         if (i > 0)
            P[i] |= (P[i-1] >> k1) & mask;
      }
   }
   reduce();
   return *this;
}
```

```
large &large::operator>>=(uint k)  // Аналогично <<=
{  int q = k / wLen, L = P.size();
   if (q >= L){*this = 0; return *this;}
   if (q)
   {  for (int i=q; i<L; i++) P[i-q] = P[i];
      SetLen(L - q);
      k %= wLen;
      if (k == 0){reduce(); return *this;}
   }
   int n = P.size() - 1, k1 = wLen - k;
   uint mask = (1 << k) - 1;
   for (int i=0; i<=n; i++)
   {  P[i] >>= k;
      if (i < n) P[i] |= ((P[i+1] & mask) << k1);
   }
   reduce();
   return *this;
}

// compare возвращает: отрицательное число, если *this < y,
// ноль, если *this == y, и положительное, если *this > y.
int large::compare(const large &y)const
{  if (neg != y.neg) return y.neg - neg;
   int code = 0, L = P.size(), Ly = y.P.size();
   if (L == 0 || Ly == 0) code = L - Ly; else
   if (L < Ly) code = -1; else
   if (L > Ly) code = +1; else
   for (int i = L - 1; i >= 0; i--)
   {  if (P[i] > y.P[i]) {code = 1; break;} else
      if (P[i] < y.P[i]) {code = -1; break;}
   }
   return neg ? -code : code;
}

// Двуцифровое произведение (Hi, Lo) = A * B:
void large::DDproduct(uint A, uint B,
                      uint &Hi, uint &Lo)const
{  uint  hiA = A >> hLen, loA = A & rMask,
         hiB = B >> hLen, loB = B & rMask,
         mid1, mid2, old;
   Lo = loA * loB; Hi = hiA * hiB;
   mid1 = loA * hiB; mid2 = hiA * loB;
   old = Lo;
   Lo += mid1 << hLen;
   Hi += (Lo < old) + (mid1 >> hLen);
   old = Lo;
   Lo += mid2 << hLen;
   Hi += (Lo < old) + (mid2 >> hLen);
}
```

```
// Двуцифровое значение (A, B) делится на d:
uint large::DDquotient(uint A, uint B, uint d)const
{  uint left, middle, right, qHi, qLo, x, dLo1,
   dHi = d >> hLen, dLo = d & rMask;
   qHi = A/(dHi + 1);
   // Это начальное приближение к qHi может оказаться
   // слишком малым.
   middle = qHi * dLo;
   left = qHi * dHi;
   x = B - (middle << hLen);
   A -= (middle >> hLen) + left + (x > B); B = x;
   dLo1 = dLo << hLen;
   // Увеличить qHi при необходимости:
   while (A > dHi || (A == dHi && B >= dLo1))
   {  x = B - dLo1;
      A -= dHi + (x > B);
      B = x;
      qHi++;
   }

   qLo = ((A << hLen) | (B >> hLen))/(dHi + 1);
   // Это начальное приближение к qLo может оказаться
   // слишком малым.
   right = qLo * dLo; middle = qLo * dHi;
   x = B - right;
   A -= (x > B);
   B = x;
   x = B - (middle << hLen);
   A -= (middle >> hLen) + (x > B);
   B = x;
   // Увеличить qLo  при необходимости:
   while (A || B >= d)
   {  x = B - d;
      A -= (x > B);
      B = x;
      qLo++;
   }
   uint result = (qHi << hLen) + qLo;
   return result == 0 && qHi > 0 ? uintmax : result;
}

// Вычесть произведение q * b из a, где a и b -
// значения длиной n цифр. Остаток a - q * b
// будет меньше, чем b, и должен быть неотрицательным.
// Последнее условие может потребовать
// уменьшения q на 1:
void large::subtractmul(uint *a, uint *b, int n,
   uint &q)const
```

```
{  uint Hi, Lo, d, carry = 0;
   int i;
   for (i=0; i<n; i++)
   {  DDproduct(b[i], q, Hi, Lo);
      d = a[i];
      a[i] -= Lo;
      if (a[i] > d) carry++;
      d = a[i + 1];
      a[i + 1] -= Hi + carry;
      carry = a[i + 1] > d;
   }
   if (carry) // q was too large
   {  q--; carry = 0;
      for (i=0; i<n; i++)
      {  d = a[i] + carry;
         carry = d < carry;
         a[i] = d + b[i];
         if (a[i] < d)
         carry = 1;
      }
      a[n] = 0;
   }
}

// Нормализация путем сдвига denom и num влево,
// так что самая левая позиция denom станет 1;
// оба операнда выражения num/denom сдвигаются
// на x битовых позиций:
bool large::normalize(large &denom, large &num,
   int &x)const
{  int r = denom.P.size() - 1;
   uint y = denom.P[r]; x = 0;
   while ((y & lBit) == 0){y <<= 1; x++;}
   denom <<= x; num <<= x;
   // Возможно второе действие согласно К. Дж. Мифсуду
   // (см. библиографию):
   if (r > 0 && denom.P[r] < denom.P[r-1])
   {  denom *= uintmax; num *= uintmax;
      return 1;
   }
   return 0;
}

// Откатить нормализацию (включая поправку Мифсуда,
// если SecondDone == 1), чтобы получить
// правильный остаток:
void large::unnormalize(large &rem, int x, bool
   SecondDone)const
```

```
         // LP10 = p10 = pow(10, ip10)
      large R, LP10 = p10;
      bool neg = x.neg;
      do
      {  x.divide(LP10, x, R, 1);
         r = (R.P.size() ? R.P[0] : 0);
         for (uint j=0; j<ip10; j++)
         {  s.push_back(char(r % 10 + '0'));
            r /= 10;
            if (r + x.P.size() == 0) break;
         }
      }  while (x.P.size());
      if (neg) s.push_back('-');
      // s содержит строку в обратном порядке
   }
}

ostream &operator<<(ostream &os, const large &x)
{  vector<char> s;
   x.num2char(s);
   vector<char>::reverse_iterator i;
   for (i=s.rbegin(); i != s.rend(); ++i) os << *i;
   return os;
}

istream &operator>>(istream &is, large &x)
{  char ch;
   x = 0;

   bool neg = 0;
   is >> ch;

   if (ch == '-')
   {  neg = 1;
      is.get(ch);
   }

   while (isdigit(ch))
   {  x = x * 10 + (ch - '0');
      is.get(ch);
   }

   if (neg) x = -x;
   is.putback(ch);
   return is;
}

large abs(large a)
{  if (a < 0) a = -a;
   return a;
}
```

```
large sqrt(const large &a)
{  large x = a, b = a, q;
   b <<= 1;
   while (b >>= 2, b > large(0)) x >>= 1;
   while (x > (q = a/x) + 1 || x < q - 1)
   {  x += q; x >>= 1;
   }
   return x < q ? x : q;
}

large power(large x, uint n)
{  large y=1;
   while (n)
   {  if (n & 1) y *= x;
      x *= x; n >>= 1;
   }
   return y;
}
```

8.3. Вычисление числа π

Хотя класс *large* предназначен для представления больших целых чисел, мы можем использовать его для приближенного представления вещественных чисел, если будем использовать соответствующее масштабирование. Давайте продемонстрируем это путем вычисления числа π с произвольной степенью точности. Один из известных способов вычисления этой константы с помощью формулы

$$\pi = 16 \arctan \frac{1}{5} - 4 \arctan \frac{1}{139}$$

Джона Мачина (1680 –1752). Вместо этого мы воспользуемся формулой

$$\pi = 48 \arctan \frac{1}{18} + 32 \arctan \frac{1}{57} - 20 \arctan \frac{1}{239} \qquad (1)$$

которая предпочтительнее, поскольку наименьший знаменатель в этой формуле (18) больше, чем в формуле Мачина (5). Математики, интересующиеся задачей вычисления числа π, могут обратиться к книге Borwein, J.M. and Borwein, P.B. (1987) *Pi and the AGM*, в которой приведены более сложные алгоритмы. Поскольку нас интересует программирование, мы просто примем приведенную формулу с тремя арктангенсами без доказательства. В свою очередь, функцию арктангенса мы будем аппроксимировать с помощью алгебраических операций путем приближенного вычисления следующего ряда:

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots \qquad (2)$$

Пользователь программы должен задать n – требуемое количество десятичных знаков после запятой. Например, в выводе программы число π будет представлено как

```
3.
1415
```

если n равно 4. Так как мы приближенно представляем вещественные числа с помощью целых, то должны обращать внимание на последние цифры результата, чтобы избежать ошибок округления, связанных с нашим приближенным вычислением функции *arctan*. Поэтому будем использовать коэффициент масштабирования

$$TenPower = 10^{n+3} \qquad (3)$$

и опустим последние три цифры при выводе результата. Вместо π мы на самом деле будем вычислять целое число, приблизительно равное $TenPower \times \pi$.

В соответствии с формулой (1) переменная x в уравнении (2) может принимать значения, равные $1/k$, где k = 18, 57 или 239. Умножая уравнение (2) на большую константу N, мы получим следующее приближение, где вторая строка содержит только целые числа:

$$N \arctan x = N \arctan (1/k) \approx$$
$$N/k - N/(3k^3) + N/(5k^5) - N/(7k^7) + \ldots = Atan(k, N) \qquad (4)$$

Здесь N является большим целым числом, а оператор деления / означает целочисленное деление (как в выражении 39/5 = 7). В результате многоточие в выражении (4) обозначает конечное количество членов $N/(ik^i)$, поэтому мы можем вычислить все эти значения. После этого из формулы (1) следует, что $TenPower \times \pi$ приблизительно равняется следующему большому числу:

$$Atan(18, 48 \times TenPower) +$$
$$Atan(57, 32 \times TenPower) -$$
$$Atan(239, 20 \times TenPower)$$

Следующая программа использует это выражение в вычислениях:

```
// largepi.cpp: Используем целые числа класса large
//              для вычисления числа Пи.
//              Скомпоновать с large.cpp.

#include <fstream>
#include <time.h>
#include <stdlib.h>
#include "large.h"
```

```
int main()
{  int n, m;
   cout << "Computation of pi. Number of decimals: ";
   cin >> n; m = n + 3;
   cout <<
      "Copy of output to file pi.txt desired (y/n)? ";
   char answer;
   cin >> answer;
   ofstream ofile;
   if (answer == 'Y' || answer == 'y')
      ofile.open("pi.txt");
   large TenPower, Pi;
   clock_t tStart, tEnd;
   tStart = clock();
   TenPower = power(5, m); TenPower <<= m;
      // Быстрее, чем TenPower = power(10, m);

   Pi    = (Atan(18, TenPower * 48)
      + Atan(57, TenPower * 32)
      - Atan(239, TenPower * 20))/1000;

   tEnd = clock();
   cout << "Digits of pi:" << endl;
   PiOutput(cout, ofile, Pi, n);
   cout << "\nTime: " << (tEnd - tStart) << " ticks\n";
   return 0;
}
```

Для *n*, существенно больших 1000, выполнение программы начинает занимать ощутимое время. Чтобы предоставить пользователю информацию о текущем состоянии расчетов, желательно показывать промежуточный вывод для каждого из следующих шагов:

1. Вычисление *TenPower* = 10^m, где $m = n + 3$.
2. Вычисление *Atan*(239, *TenPower* * 20).
3. Вычисление *Atan*(57, *TenPower* * 32).
4. Вычисление *Atan*(18, *TenPower* * 48).
5. Перевод переменной *Pi* класса *large* в символьную строку, содержащую *n* десятичных цифр для вывода.

На шаге 1 мы оптимизируем наши расчеты с помощью операции сдвига влево, заметив, что $10^m = 5^m \times 2^m$. Отсюда мы можем вычислить значение 5^m, а затем сдвинуть результат на *m* двоичных позиций влево. Поскольку время вычисления произведения чисел класса *large* зависит от их длины, мы можем вычислить значение 5^m быстрее, чем 10^m.

Шаги 2, 3 и 4 необязательно выполнять в этом порядке. Когда требуется вычислить значение выражения *f*() – *g*(), язык C++ не указывает, в каком порядке происходят вызовы *f* и *g*. В нашем примере *g* вычисляется раньше, чем *f*.

После того как пользователь введет требуемое количество десятичных знаков, он должен указать, требуется ли записывать вывод программы в текстовый файл *pi.txt*. Затем программа показывает, что она выполняет шаги 2, 3 и 4, выводя на экран соответствующую информацию, как видно из примера на следующей странице.

Приводимый пример выполнялся на компьютере Pentium 166 с 32 Мб памяти; программа была откомпилирована с помощью BC5 и запущена в окне DOS операционной системы Windows 95. Поскольку для BC5 интервал таймера (tick) равен 1 миллисекунде, вычисление значения π в этом примере с точностью до 1000 десятичных знаков заняло около 0.3 секунды (не считая времени, потраченного на вывод текста на экран). Результаты проверялись путем сравнения десяти последних цифр этого результата (5 493 624 646) с результатами Ясамасы Канады из Токио, который вычислил миллиард знаков числа π с помощью совершенно другого аппаратного и программного обеспечения.

Как упоминалось в начале этой главы, моя предыдущая.

книга *Algorithms and Data Structures in C++* содержит класс *large*, аналогичный представленному здесь, но не использующий STL. Вычисление π с помощью предыдущей версии программы заняло примерно такое же время, как и у представленной здесь версии, использующей STL. Однако на программирование класса *large* без STL разработчиком было затрачено гораздо больше времени. Несколько преувеличивая, мы можем сказать, что между программированием с применением STL и без примерно такая же разница, как у программирования на языках высокого уровня с программированием на ассемблере. Хотя программирование на ассемблере дает более эффективный код, в наше время использование ассемблера абсолютно нереально для большинства приложений. Программисты, отвергающие STL из-за боязни уменьшения эффективности, похожи на тех людей, которые в 60-е годы считали, что любая серьезная задача должна быть запрограммирована с помощью ассемблера[1].

[1] Часто программы, использующие STL, выполняются даже эффективнее программ, которые используют другие методы. Так, например, алгоритм *sort*, определенный в STL, в целом несколько эффективнее, чем алгоритм *qsort*, определенный в стандартной библиотеке языка C. – *Прим. переводчика.*

```
Computation of pi. Number of decimals: 1000
Copy of output to file pi.txt desired (y/n)? n
k = 239
k = 57
k = 18
Digits of pi:
3.
1415926535 8979323846 2643383279 5028841971 6939937510
5820974944 5923078164 0628620899 8628034825 3421170679
8214808651 3282306647 0938446095 5058223172 5359408128
4811174502 8410270193 8521105559 6446229489 5493038196
4428810975 6659334461 2847564823 3786783165 2712019091
4564856692 3460348610 4543266482 1339360726 0249141273
7245870066 0631558817 4881520920 9628292540 9171536436
7892590360 0113305305 4882046652 1384146951 9415116094
3305727036 5759591953 0921861173 8193261179 3105118548
0744623799 6274956735 1885752724 8912279381 8301194912
9833673362 4406566430 8602139494 6395224737 1907021798
6094370277 0539217176 2931767523 8467481846 7669405132
0005681271 4526356082 7785771342 7577896091 7363717872
1468440901 2249534301 4654958537 1050792279 6892589235
4201995611 2129021960 8640344181 5981362977 4771309960
5187072113 4999999837 2978049951 0597317328 1609631859
5024459455 3469083026 4252230825 3344685035 2619311881
7101000313 7838752886 5875332083 8142061717 7669147303
5982534904 2875546873 1159562863 8823537875 9375195778
1857780532 1712268066 1300192787 6611195909 2164201989

Time: 312 ticks
```

Указатель идентификаторов и английских названий

I

L

M

N

O

P

Q

R

S

Предметный указатель

Леен Аммерааль

STL для программистов C++

Главный редактор Мовчан Д. А.
Перевод Баранов Ю.А.
Литературный редактор Космачева Н.А.
Технический редактор Волнов Ю.А.
Верстка Али-Заде В.Х.
Дизайн обложки Кудряшев А.В.

Гарнитура «Петербург». Печать офсетная
Усл. печ. л. 15. Тираж 3000. Зак. №

Отпечатано в полном соответствии
с качеством предоставленных диапозитивов

STL для программистов на C++

"Это первая книга из тех, что я прочел, которая позволяет профессиональному программисту быстро начать использовать STL."

Фрэнсис Глассборо,
Председатель Ассоциации Пользователей C и C++ (ACCU)

Стандартная библиотека шаблонов (STL) содержит множество полезных инструментов общего назначения.
В этой книге наряду со справочным материалом последовательно изложено введение в предмет, которое позволит вам быстро освоить основы применения STL в программировании. Небольшие законченные программы служат иллюстрацией основных понятий STL, таких как контейнеры, алгоритмы и функциональные объекты. В книге имеется раздел, посвященный новому классу string.
Все алгоритмы STL сначала представляются формально в виде прототипа, а затем следует неформальное объяснение, как применять их на практике. Все понятия иллюстрируются большим количеством примеров программ. И наконец, приводятся специальные примеры для облегчения понимания нетривиальных понятий, таких как функциональные объекты и адаптеры функций, включая предикаты, привязки и отрицатели.

Internet-магазин:
www.alians-kniga.ru
Книга - почтой:
Россия, 123242, Москва, а/я 20
e-mail: orders@alians-kniga.ru
Оптовая продажа:
"Альянс-книга"
(495)258-9194, 258-9195
e-mail: books@alians-kniga.ru

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

ISBN 5-89818-027-3
9 785898 180270