Глава 14

Библиотека стандартных шаблонов.

Поздравляем! Если при изучении предыдущих глав этой книги вы действительно работали, то теперь можете с полным правом называть себя состоявшимся программистом на C++. В этой последней главе мы расскажем об одном из наиболее увлекательных и совершенных инструментов языка программирования C++ - библиотеке стандартных шаблонов (Standard Template Library, STL).

Библиотека стандартных шаблонов не являлась частью исходной спецификации C++, а была добавлена к ней позже, в процессе стандартизации, на что и были направлены основные усилия разработчиков. Библиотека стандартных шаблонов обеспечивает общецелевые, стандартные классы и функции, которые реализуют наиболее популярные и широко используемые алгоритмы и структуры данных. Например, в библиотеке стандартных шаблонов поддерживаются уже известные нам векторы (vectors), списки (lists), очереди (queues) и стеки (stacks). В ней также определены различные процедуры доступа к этим структурам данных. Поскольку библиотека стандартных шаблонов строится на основе классов-шаблонов, входящие в неё алгоритмы и структуры применимы почти ко всем типам данных.

Рассказ о библиотеке стандартных шаблонов необходимо начать с признания того факта, что она представляет собой вершину искусства программирования, и в ней используются самые изощрённые свойства C++. Чтобы научиться понимать и применять библиотеку стандартных шаблонов, вам следует досконально освоить материал предыдущих глав и уметь свободно оперировать полученными знаниями. В особенности это касается шаблонов. Синтаксис шаблонов, на которых написана библиотека стандартных шаблонов, может показаться совершенно устрашающим, но не надо бояться, он выглядит сложнее, чем это есть на самом деле. Помните, в этой главе нет ничего более сложного, чем то, с чем вы уже познакомились в предыдущих главах книги, поэтому не надо расстраиваться или пугаться, если на первых порах библиотека стандартных шаблонов покажется вам непонятной. Немного терпения, усидчивости, экспериментов и главное, не позволяйте незнакомому синтаксису заслонить от вас исходную простоту библиотеки стандартных шаблонов.

Библиотека стандартных шаблонов достаточно велика, поэтому вы узнаете здесь далеко не обо всех её свойствах. Фактически, полного описания библиотеки, всех её свойств, нюансов и приёмов программирования хватило бы на большую отдельную книгу. Представленный в этой главе обзор предназначен для того, чтобы познакомить вас с её базовыми операциями, философией, основами программирования. После усвоения этого материала вы, несомненно, легко сможете проделать оставшуюся часть пути самостоятельно.

Помимо библиотеки стандартных шаблонов в этой главе описан один из наиболее важных новых классов C++ - *строковый класс (string class)*. Строковый класс определяет строковый тип данных, что позволяет работать с символьными строками почти так же, как это делается с данными других типов.

14.1. Знакомство с библиотекой стандартных шаблонов.

Хотя библиотека стандартных шаблонов достаточно велика, а её синтаксис иногда пугающе сложен, с ней гораздо проще работать, если понять, как она образована и из каких элементов состоит. Поэтому перед изучением примеров программ вполне оправдано дать её краткий обзор.

Ядро библиотеки стандартных шаблонов образуют три основополагающих элемента: контейнеры, алгоритмы и итераторы. Эти элементы функционируют в тесной взаимосвязи друг с другом, обеспечивая искомые решения проблем программирования.

Контейнеры (containers) – это объекты, предназначенные для хранения других объектов. Контейнеры бывают различных типов. Например, в классе **vector** (вектор) определяется динамический массив, в классе **queue** (очередь) – очередь, в классе **list** (список) – линейный список. Помимо базовых контейнеров, в библиотеке стандартных шаблонов определены также *ассоциативные контейнеры (associative containers)*, позволяющие с помощью ключей (keys) быстро получать хранящиеся в них значения. Например, в классе **map** (ассоциативный список) определяется ассоциативный список, обеспечивающий доступ к значениям по уникальным ключам. То есть, в ассоциативных списках хранятся пары величин ключ/значение, что позволяет при наличии ключа получить соответствующее ключу значение.

В каждом классе-контейнере определяется набор функций для работы с этим контейнером. Например, список содержит функции для вставки, удаления и слияния (merge) элементов. В стеке имеются функции для размещения элемента в стеке и извлечения его из стека.

Алгоритмы (algorithms) выполняют операции над содержимым контейнеров. Существуют алгоритмы для инициализации, сортировки, поиска и замены содержимого контейнеров. Многие алгоритмы предназначены для работы с *последовательностью (sequence)*, которая представляет собой линейный список элементов внутри контейнера.

Итераторы (iterators) – это объекты, которые по отношению к контейнерам играют роль указателей. Они позволяют получать доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива. Имеется пять типов итераторов, которые описаны ниже:

Итератор	**Описание**
Произвольного доступа (random access)	Используется для считывания и записи значений. Доступ к элементам произвольный
Двунаправленный (bidirectional)	Используется для считывания и записи значений. Может проходить контейнер в обоих направлениях.
Однонаправленный (forward)	Используется для считывания и записи значений. Может проходить контейнер только в одном направлении.
Ввода (input)	Используется только для считывания значений. Может проходить контейнер только в одном направлении.
Вывода (output)	Используется для записи значений. Может проходить контейнер только в одном направлении.

(Не запутайтесь. По аналогии с потоковым вводом/выводом под *вводом* понимается ввод информации *из* контейнера, т.е. считывание, а под *выводом* – вывод информации *в* контейнер, т.е. запись – *примеч.пер.*)

Как правило, итератор с большими возможностями доступа к содержимому контейнера может использоваться вместо итератора с меньшими возможностями. Например, однонаправленным итератором можно заменить итератор ввода.

С итераторами можно работать точно так же, как с указателями. Над ними можно выполнять операции инкремента и декремента. К ним можно применить оператор *****. Типом итераторов объявляется тип **iterator**, который определён в различных контейнерах.

В библиотеке стандартных шаблонов также поддерживаются *обратные итераторы (reverse iterators)*. Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, но проходящие последовательность в обратном направлении. То есть, если обратный итератор указывает на последний элемент последовательности, то инкремент этого итератора приведёт к тому, что он будет указывать на элемент перед последним.

При упоминании различных типов итераторов в описаниях шаблонов, в данной книге будут использоваться следующие термины:

Термин	Тип итератора
Randlter	Произвольного доступа (random access)
Bilter	Двунаправленный (bidirectional)
Forlter	Однонаправленный (forward)
Inlter	Ввода (input)
Oulter	Вывода (output)

Вдобавок к контейнерам, алгоритмам и итераторам, в библиотеке стандартных шаблонов поддерживается ещё несколько стандартных компонентов. Главными среди них являются распределители памяти, предикаты и функции сравнения.

У каждого контейнера имеется определённый для него *распределитель памяти (allocator)*, который управляет процессом выделения памяти для контейнера. По умолчанию распределителем памяти является объект класса **allocator**, но вы можете определить собственный распределитель памяти, если хотите возложить на него какие-нибудь необычные функции. В большинстве случаев достаточно распределителя памяти, заданного по умолчанию.

В некоторых алгоритмах и контейнерах используется функция особого вида, называемая *предикатом (predicate)*. Предикат может бинарным или унарным. У унарного предиката один аргумент, а у бинарного – два. Возвращаемым значением этих функций является значение истина или ложь. Точные условия получения того или иного значения определяются программистом. Все унарные предикаты, которые будут упомянуты в этой главе, имеют тип **UnPred**, а все бинарные – **BinPred**. Аргументы бинарного предиката всегда расположены по порядку: *первый, второй*. Тип аргументов как унарного, так и бинарного предиката соответствует типу хранящихся в контейнере объектов.

В некоторых алгоритмах и классах используется специальный тип бинарного предиката, предназначенный для сравнения двух элементов. Такой предикат называется *функцией*

сравнения (comparison function). Функция сравнения возвращает истину, если её первый аргумент меньше второго. Типом функции сравнения является тип **Comp**.

Помимо заголовков для разнообразных классов-контейнеров. Входящих в библиотеку стандартных шаблонов, стандартная библиотека C++ включает также заголовки **<utility>** и **<functional>**, предназначенные для поддержки классов-шаблонов. Например, заголовочный файл **<utility>** содержит определение класса-шаблона **pair** (пара), в котором могут храниться пары значений. Позднее в этой главе мы ещё воспользуемся шаблоном **pair**.

Шаблоны из заголовочного файла **<functional>** помогают создавать объекты, определяющие оператор-функцию **operator()**. Эти объект называются *объектами-функциями (function objects)* и во многих случаях могут использоваться вместо указателей на функцию. В заголовочном файле **<functional>** объявлено несколько встроенных объектов-функций, некоторые из которых перечислены ниже:

plus	divides	equal_to	greater_equal	logical_and
minus	modulus	not_equal_to	less	logical_or
multiplies	negate	greater	less_equal	logical_not

Вероятно, чаще других применяется объект-функция **less** (меньше), которая позволяет определить, является ли значение одного объекта меньше, чем значение другого. В описываемых далее алгоритмах библиотеки стандартных шаблонов объектами-функциями можно заменять указатели на реальные функции. Если использовать объекты-функции вместо указателей на функцию, библиотека стандартных шаблонов будет генерировать более эффективный код. Тем не менее для целей данной главы (обзор библиотеки стандартных шаблонов) объекты-функции не нужны и непосредственно применяться не будут. Хотя сами по себе объекты-функции не представляют особой сложности, их подробное обсуждение достаточно продолжительно и выходит за рамки нашей книги. Этот материал вам следует освоить самостоятельно, если в будущем вы захотите использовать библиотеку стандартных шаблонов с максимальной эффективностью.

14.2. Классы-контейнеры.

Ранее уже объяснялось, что контейнерами называются объекты библиотеки стандартных шаблонов, непосредственно предназначенные для хранения данных. В табл. 14.1 перечислены контейнеры, определённые в библиотеке стандартных шаблонов, а также заголовки, которые следует включить в программу, чтобы использовать тот или иной контейнер. Хотя строковый класс, который управляет символьными строками, также является контейнером, ему будет посвящён отдельный раздел.

Таблица 14.1 *Контейнеры, определённые в библиотеке стандартных шаблонов.*

Контейнер	Описание	Заголовок
bitset	Множество битов	<bitset>
deque	Двусторонняя очередь	<deque>
list	Линейный список	<list>
map	Ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано только одно значение	<map>
multimap	Ассоциативный список для хранения пар	<map>

	ключ/значение, где с каждым ключом связано два или более значений	
multiset	Множество, в котором каждый элемент не обязательно уникален	<set>
priority_queue	Очередь с приоритетом	<queue>
queue	Очередь	<queue>
set	Множество, в котором каждый элемент уникален	<set>
stack	Стек	<stack>
vector	Динамический массив	<vector>

Поскольку имена типов элементов, входящих в объявление класса-шаблона, могут быть самыми разными, в классах-контейнерах с помощью ключевого слова **typedef** объявляются некоторые согласованные версии этих типов. Ниже представлены имена типов, конкретизированные с помощью ключевого слова **typedef**, которые можно встретить чаще других.

Согласованное имя типа	Описание
size_type	Интегральный тип, эквивалентный типу **size_t**
reference	Ссылка на элемент
const_reference	Постоянная ссылка на элемент
iterator	Итератор
const_iterator	Постоянный итератор
reverse_iterator	Обратный итератор
const_reverse_iterator	Постоянный обратный итератор
value_type	Тип хранящегося в контейнере значения
allocator_type	Тип распределителя памяти
key_type	Тип ключа
key_compare	Тип функции, которая сравнивает два ключа
value_compare	Тип функции, которая сравнивает два значения

Хотя изучить все контейнеры в рамках одной главы невозможно, в следующих разделах рассказывается о трёх из них: векторе, списке и ассоциативном списке. Если вы поймёте, как работают эти три контейнера, то с другими классами библиотеки стандартных шаблонов у вас проблем не будет.

14.3. Векторы.

Вероятно, самым популярным контейнером является вектор. В классе **vector** поддерживаются динамические массивы. Динамическим массивом называется массив, размеры которого могут увеличиваться по мере необходимости. Как известно, в C++ в процессе компиляции размеры массива фиксируются. Хотя это наиболее эффективный способ реализации массивов, одновременно он и самый ограниченный, поскольку не позволяет адаптировать размер массива к изменяющимся в процессе выполнения программы условиям. Решает проблему вектор, который выделяет память для массива по мере возникновения потребности в этой памяти. Несмотря на то, что вектор является, по сути, динамическим массивом, для доступа к его элементам подходит обычная индексная нотация, которая используется для доступа к элементам стандартного массива.

Ниже представлена спецификация шаблона для класса **vector**:

template <class T, class Allocator = allocator <T>>class vector

Здесь **T** – это тип предназначенных для хранения в контейнере данных, а ключевое слово **Allocator** задаёт распределитель памяти, который по умолчанию является стандартным распределителем памяти. В классе **vector** определены следующие конструкторы:

explicit vector (const Allocator &a = Allocator());

explicit vector (size_type *число*, const T & *значение* = T (),
const Allocator &a = Allocator ());

vector (const vector<T, Allocator> &*объект*);

template<class InIter>vector(InIter *начало*, InIter *конец*,
const Allocator &a = Allocator());

Первая форма представляет собой конструктор пустого вектора. Во второй форме конструктора вектора число элементов – это ***число***, а каждый элемент равен значению ***значение***. Параметр ***значение*** может быть значением по умолчанию. В третьей форме конструктора вектор предназначен для одинаковых элементов, каждый из которых – это ***объект***. Четвёртая форма – это конструктор вектора, содержащего диапазон элементов, заданный итераторами ***начало*** и ***конец***.

Для любого объекта, который будет храниться в векторе, должен быть определён конструктор по умолчанию. Кроме этого, для объекта должны быть определены операторы < и ==. Для некоторых компиляторов может потребоваться определить и другие операторы сравнения. (Для получения более точной информации обратитесь к документации на ваш компилятор.) Для встроенных типов данных все указанные требования выполняются автоматически.

Хотя синтаксис выглядит довольно сложно, в объявлении вектора ничего сложного нет. Ниже представлено несколько примеров такого объявления:

vector<int> iv; // создание вектора нулевой длины для целых

vector<char> cv(5); // создание пятиэлементного вектора для символов

vector<char> cv(5, ‘x’); // создание и инициализация
// пятиэлементного вектора для символов

vector<int> iv2(iv); // создание вектора для целых
// из вектора для целых

Для класса **vector** определяются следующие операторы сравнения:

==, <, <=, !=, >, >=

Кроме этого для класса **vector** определяется оператор индекса **[]**, что обеспечивает доступ к элементам вектора посредством обычной индексной нотации, которая используется для доступа к элементам стандартного массива.

В табл. 14.2 представлены функции – члены класса **vector**. (Повторяем, не нужно пугаться необычного синтаксиса.) наиболее важными фнкциями-членами являются функции **size(), begin(), end(), push_back(), insert()** и **erase**. Функция **size()** возвращает текущий размер вектора. Эта функция особенно полезна, поскольку позволяет узнать размер вектора во время выполнения программы. Помните, вектор может расти по мере необходимости, поэтому размер вектора необходимо определять не в процессе компиляции, а в процессе выполнения программы.

Функция **begin()** возвращает итератор начала вектора. Функция **end()** возвращает итератор конца вектора. Как уже говорилось, итератора очень похожи на указатели и с помощью функций **begin()** и **end()** можно получить итераторы (читай: указатели) начала и конца вектора.

Функция **push_back()** помещает значение в конец вектора. Если это необходимо для размещения нового элемента, вектор удлиняется. В середину вектора элемент можно добавить с помощью функции **insert()**. Вектор можно инициализировать. В любом случае, если в векторе хранятся элементы, то с помощью оператора индекса массива к этим элементам можно получить доступ и их изменить. Удалить элементы из вектора можно с помощью функции **erase**.

Таблица 14.2 *Функции-члены класса* ***vector.***

Функция-член	**Описание**
template<class Inlter> **void assign (Inlter *начало,* Inlter *конец*);**	Присваивает вектору последовательность, определённую итераторами ***начало*** и ***конец***.
template<class Size, class T> **void assign (size *число,*** **const T &*значение* =T**	Присваивает вектору ***число*** элементов, причём значение каждого элемента равно параметру ***значение***.
reference at (size_type *i*); **const_reference** **at(size_type *i*) const;**	Возвращает ссылку на элемент, заданный параметром ***i***.
reference back(); **const_reference back{) const;**	Возвращает ссылку на последний элемент вектора.
iterator begin(); **const_iterator begin() const;**	Возвращает итератор первого элемента вектора.
size_type capacity() const;	Возвращает текущую емкость вектора, т. е. то число элементов, которое можно разместить в векторе без необходимости выделения дополнительной области памяти.
void clear();	Удаляет все элементы вектора.
bool empty() const;	Возвращает истину, если вызывающий вектор ***i*** пуст, в противном случае возвращает ложь.
iterator end(); **const_iterator end() const;**	Возвращает итератор конца вектора.
iterator erase(iterator *i*);	Удаляет элемент, на который указывает итератор ***i***. Возвращает итератор элемента, который расположен следующим за удаленным.
iterator erase (iterator *начало,* iterator *конец);*	Удаляет элементы, заданные между итераторами ***начало*** **и** ***конец.*** Возвращает

	итератор элемента, который расположен следующим за последним удаленным.
reference front(); **const_reference front() const;**	Возвращает ссылку на первый элемент вектора.
allocator_type get_allocator() const;	Возвращает распределитель памяти вектора.
iterator insert(iterator /, **const T *&значение* = T());**	Вставляет параметр ***значение*** перед элементом, заданным итератором ***i***. Возвращает итератор элемента.
void insert(iterator /, **size_type *число,* const T *&значение);***	Вставляет ***число*** копий параметра ***значение*** л перед элементом, заданным итератором ***i***.
template<class lnlter> **void insert(iterator *i*, Inlter *начало,* Inlter *конец);***	Вставляет последовательность, определенную между итераторами ***начало*** **и** ***конец,*** перед элементом, заданным итератором ***i.***
size_type max_size() const;	Возвращает максимальное число элементов, которое может храниться в векторе.
reference operator[] **(size_type /) const;** **const_reference operator!]** **(size_type /) const;**	Возвращает ссылку на элемент, заданный параметром ***i.***
void pop_back();	Удаляет последний элемент вектора.
void push_back(const T *&значение);*	Добавляет в конец вектора элемент, значение которого равно параметру ***значение.***
reverse_iterator rbegin(); **const_reverse_iterator rbegin() const;**	Возвращает обратный итератор конца вектора.
reverse_iterator rend(); **const_reverse_iterator rend() const;**	Возвращает обратный итератор начала вектора.
void reserve (size _type число);	Устанавливает емкость вектора равной, по меньшей мере, параметру ***число*** элементов.
void resize(size_type *число,* **T *значение* = T());**	Изменяет размер вектора в соответствии с параметром ***число.*** Если при этом вектор удлиняется, то добавляемые в конец вектора элементы получают значение, заданное параметром ***значение.***
size_type size() const;	Возвращает хранящееся на данный момент в векторе число элементов.
void swap(vector<T, **Allocator> *&объект);***	Обменивает элементы, хранящиеся в вызывающем векторе, с элементами в объекте ***объект.***

Примеры

1. В представленном ниже коротком примере показаны основные операции, которые можно выполнять при работе с вектором.

```
// Основные операции вектора
```

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

int main()
{
   vector<int> v; // создание вектора нулевой длины
   int i;

   // вывод на экран размера исходного вектора v
   cout << "Размер = " << v.size() << endl;

   // помещение значений в конец вектора,
   // по мере необходимости вектор будет расти
   for(i=0; i<10; i++) v.push_back(i);

   // вывод на экран текущего размера вектора v
   cout << "Новый размер = " << v.size() << endl;

   // вывод на экран содержимого вектора v
   cout << "Текущее содержимое:\n";
   for(i=0; i<v.size(); i++) cout << v[i] << " ";
   cout << endl;

   // помещение новых значений в конец вектора,
   // и опять по мере необходимости вектор будет расти
   for(i=0; i<10; i++) v.push_back(i+10);

   // вывод на экран текущего размера вектора
   cout << "Новый размер = " << v.size() << endl;

   // вывод на экран содержимого вектора
   cout << "Текущее содержимое:\n";
   for(i=0; i<v.size(); i++) cout << v[i] << " ";
   cout << endl;

   // изменение содержимого вектора
   for(i=0; i<v.size(); i++) v[i] = v[i] + v[i];

   // вывод на экран содержимого вектора
   cout << "Удвоенное содержимое:\n";
   for(i=0; i<v.size(); i++) cout << v[i] << " ";
   cout << endl;

   return 0;
}
```

После выполнения программы на экране появится следующее:

```
Размер =0
Новый размер = 10
Текущее содержимое:
0 1 2 3 4 5 6 7 8 9
Новый размер = 20
Текущее содержимое:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Удвоенное содержимое:
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

Тщательно проанализируйте программу. В функции **main()** создаётся вектор **v** для хранения целых. Поскольку не используется никакой инициализации, это пустой вектор с равной нулю начальной ёмкостью, то есть это вектор нулевой длины. Этот факт подтверждается

вызовом функции-члена **size()**. Далее с помощью функции-члена **push_back()** к концу вектора **v** добавляется десять элементов. Чтобы разместить эти новые элементы, вектор **v** вынужден увеличиться. Как показывает выводимая информация, его размер стал равным 10. После этого выводится содержимое вектора **v**. Обратите внимание, что для этого используется обычный оператор индекса массива. Далее к вектору добавляется ещё десять элементов и , чтобы их разместить, вектор снова автоматически увеличивается. В конце концов, с помощью стандартного оператора индекса массива меняются значения элементов вектора.

В программе есть ещё кое-что интересное. Отметьте, что функция **v.size()** указана прямо в инструкции организации цикла вывода на экран содержимого вектора **v.** Одним из преимуществ векторов по сравнению с массивами является то, что вы всегда имеете возможность определить текущий размер вектора. Очевидно, что такая возможность может оказаться полезной в самых разных ситуациях.

2. Как вы знаете, в C++ массивы и указатели очень тесно связаны. Доступ к массиву можно получить либо через оператор индекса, либо через указатель. По аналогии с этим в библиотеке стандартных шаблонов имеется тесная связь между векторами и итераторами. Доступ к членам вектора можно получить либо через оператор, либо через итератор. В следующем примере показаны оба этих подхода.

```
// Организация доступа к вектору с помощью итератора
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> v; // создание вектора нулевой длины
   int i;

   // помещение значений в вектор
   for(i=0; i<10; i++) v.push_back(i);

   // доступ к содержимому вектора
   // с использованием оператора индекса
   for(i=0; i<10; i++) cout << v[i] << " ";
   cout << endl;

   // доступ к вектору через итератор
   vector<int>::iterator p = v.begin();
   while(p != v.end()) {
      cout << *p << " ";
      p++;
   }

   return 0;
}
```

После выполнения программы на экране появится следующее:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

В этой программе тоже сначала создаётся ветор **v** нулевой длины. Далее с помощью функции-члена **push_back()** к концу вектора **v** добавляются некоторые значения и размер вектора **v** увеличивается.

Обратите внимание на объявление итератора **p**. Тип **iterator** определяется с помощью класса-контейнера. То есть, чтобы получить итератор для выбранного контейнера, объявить его нужно именно так, как показано в примере: просто укажите перед типом **iterator** имя контейнера. С помощью функции-члена **begin()** итератор инициализируется, указывая на начало вектора. Теперь, применяя к итератору оператор инкремента, можно получить доступ к любому выбранному элементу вектора. Этот процесс совершенно аналогичен использованию указателя для доступа к элементам массива. С помощью функции-члена **end()** определяется факт достижения конца вектора. Возвращаемым значением этой функции является итератор того места, которое находится сразу за последним элементом вектора. Таким образом, если итератор **p** равен возвращаемому значению функции **v. end()**, значит, конец вектора был достигнут.

3. Помимо возможности размещения элементов в конце вектора, с помощью функции-члена **insert()** их можно вставлять в его середину. Удалять элементы из вектора можно с помощью функции-члена **erase**.

```
// Демонстрация функций inset() и erase()
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> v(5, 1); // создание пятиэлементного вектора
                        // из единиц
   int i;

   // вывод на экран исходных размера и содержимого вектора
   cout << "Размер = " << v.size() << endl;
   cout << "Исходное содержимое:\n";
   for(i=0; i<v.size(); i++) cout << v[i] << " ";
   cout << endl;

   vector<int>::iterator p = v.begin();
   p += 2; // указывает на третий элемент

   // вставка в вектор на то место,
   // куда указывает итератор p десяти новых элементов,
   // каждый из которых равен 9
   v.insert(p, 10, 9);

   // вывод на экран размера
   // и содержимого вектора после вставки
   cout << "Размер после вставки = " << v.size() << endl;
   cout << "Содержимое после вставки:\n";
   for(i=0; i<v.size(); i++) cout << v[i] << " ";
   cout << endl;

   // удаление вставленных элементов
   p = v.begin();
   p += 2; // указывает на третий элемент
   v.erase(p, p+10); // удаление следующих десяти элементов
                     // за элементом, на который указывает
                     // итератор p

   // вывод на экран размера
   // и содержимого вектора после удаления
   cout << "Размер после удаления = " << v.size() << endl;
   cout << "Содержимое после удаления:\n";
```

```
   for(i=0; i<v.size(); i++) cout << v[i] << " ";
   cout << endl;

   return 0;
}
```

После выполнения программы на экране появится следующее:

```
Размер = 5
Исходное содержимое:
1 1 1 1 1

Размер поля вставки =15
Содержимое поля вставки:
1 1 9 9 9 9 9 9 9 9 9 9 1 1 1

Размер после удаления = 5
Содержимое после удаления:
1 1 1 1 1
```

4. В следующем пример вектор используется для хранения объектов класса, определённого программистом. Обратите внимание, что в классе определяются конструктор по умлчанию и перегруженные версии операторов < и ==. Помните, в зависимости от того, как реализована библиотека стандартных шаблонов для вашего компилятора, вам может понадобиться определить и другие операторы сравнения.

```
// Хранение в векторе объектов пользовательского класса
#include <iostream>
#include <vector>
using namespace std;

class Demo {
   double d;
public:
   Demo() { d = 0.0; }
   Demo(double x) { d = x; }

   Demo &operator=(double x) {
      d = x; return *this;
   }
   double getd() {returned d; }
};

boll operator<(Demo a, Demo b)
{
   return a.getd() < b.getd();
}

boll operator==(Demo a, Demo b)
{
   return a.getd() == b.getd();
}

int main()
{
   vector<Demo> v;
   int i;

   for(i=0; i<10; i++)
      v.push_back(Demo(i/3.0));
```

```
    for(i=0; i<v.size(); i++)
      cout << v[i].getd() << " ";

    cout << endl;

    for(i=0; i<v.size(); i++)
      v[i] = v[i].getd() * 2.1;

    for(i=0; i<v.size(); i++)
      cout << v[i].getd() << " ";

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
0 0.333333 0.666667 1 1.33333 1.66667 2 2.33333 2.66667 3
0 0.7 1.4 2.1 2.8 3.5 4.2 4.9 5.6 6.3
```

14.4. Списки.

Класс **list** поддерживает двунаправленный линейный список. В отличии от вектора, в котором реализован произвольный доступ, к элементам списка доступ может быть только последовательным. Поскольку списки являются двунаправленными, доступ к элементам списка возможен с обеих его сторон.

Ниже представлена спецификация шаблона для класса **list:**

template<class T, class Allocator = allocator<T>>class list

Здесь **T** — это тип данных, предназначенных для хранения в списке, а ключевое слово **Allocator** задает распределитель памяти, который по умолчанию является стандартным распределителем памяти. В классе **list** определены следующие конструкторы:

explicit list(const Allocator &a = Allocator()) ;

explicit list(size type *число,* const T *&значение* = T(), const Allocator &a = Allocator()) ;

list (const list<T, Allocator>&*объект*) ; ,

teinplate<class Inlter>list(lnlter *начало,* Inlter конец, const Allocator &a = Allocator()) ;

Первая форма представляет собой конструктор пустого списка. Вторая форма — конструктор списка, число элементов которого — это ***число,*** *а* каждый элемент равен значению ***значение,*** которое может быть значением по умолчанию. Третья форма конструктора предназначена для списка из одинаковых элементов, каждый из которых — это ***объект.*** Четвертая форма — это конструктор списка, содержащего диапазон элементов, заданный итераторами ***начало*** **и** ***конец.***

Для класса **list** определяются следующие операторы сравнения:

==, <, <=, !=, >, >=

В табл. 14.3 представлены функции — члены класса **list.** Размещать элементы в конце списка можно с помощью функции **push_back()** (как и в случае с вектором), в начале — с помощью функции **push_front(),** а в середине — с помощью функции **insertO.** Для ***соединения (join)*** двух списков нужна функция **splice(),** а для ***слияния (merge)*** — функция **merge().**

Для любого типа данных, которые вы собираетесь хранить в списке, должен быть определен конструктор по умолчанию. Кроме этого, необходимо определить различные операторы сравнения. К моменту написания этой книги точные требования к объектам, предназначенным для хранения в списке, у разных компиляторов были разными, поэтому перед использованием списка тщательно изучите техническую документацию на ваш компилятор.

Таблица 14.3. *Функции — члены класса* ***list***

Функция-член	Описание
template<class lnlter> **void assign(lnlter *начало,* Inlter конец);**	Присваивает списку последовательность, определенную итераторами ***начало* и *конец.***
template<class Size, class T> **void assign(Size *число,*** **const T *Означение* = T());**	Присваивает списку ***число*** элементов, причем значение каждого элемента равно параметру ***значение.***
reference back(); **const_reference back() const;**	Возвращает ссылку на последний элемент списка.
iterator begin(); **const_iterator begin() const;**	Возвращает итератор первого элемента списка.
void clear();	Удаляет все элементы списка.
bool empty() const;	Возвращает истину, если вызывающий список пуст, в противном случае возвращает ложь.
iterator end(); **const_iterator end() const;**	Возвращает итератор конца списка.
iterator erase (iterator *i*);	Удаляет элемент, на который указывает итератор ***i***. Возвращает итератор элемента, который расположен следующим за удаленным.
iterator erase (iterator *начало,* iterator *конец);*	Удаляет элементы, заданные между итераторами ***начало* и *конец.*** Возвращает итератор элемента, который расположен следующим за последним удаленным.
reference front(); **const_reference front() const;**	Возвращает ссылку на первый элемент списка.
allocator_type **get_allocator() const;**	Возвращает распределитель памяти списка.
iterator insert(iterator *i,* **const T *&значение* = T());**	Вставляет параметр ***значение*** перед элементом, заданным итератором ***i***. Возвращает итератор элемента.
void insert(lterator *i,* **size_type *число,* const T &значение);**	Вставляет ***число*** копий параметра ***значение*** перед элементом, заданным итератором ***i.***
template<class lnlter> **void insert(iterator *i,**	Вставляет последовательность, определенную между итераторами ***начало* и**

Inlter *начало,* Inlter *конец);*	***конец,*** перед элементом, заданным итератором ***i.***
size_type max_size() const;	Возвращает максимальное число элементов, которое может храниться в списке.
void merge(list<T, Allocator> *&объект);* **template<class Comp>** **void merge(list<T, Allocator> *&объект,*** **Comp *ф_сравн);***	Выполняет слияние упорядоченного списка, хранящегося в объекте ***объект,*** *с* вызывающим упорядоченным списком. Результат упорядочивается. После слияния список, хранящийся в объекте ***объект*** становится пустым. Во второй форме для определения того, является ли значение одного элемента меньшим, чем значение другого, может задаваться функция сравнения ***ф_сравн.***
void pop_back();	Удаляет последний элемент списка.
void pop_front();	Удаляет первый элемент списка.
void push_back(const T &значение);	Добавляет в конец списка элемент, значение которого равно параметру ***значение.***
void push_front(const T &значение);	Добавляет в начало списка элемент, значение которого равно параметру ***значение.***
reverse_iterator rbegin(); **const_reverse_iterator rbegin() const;**	Возвращает обратный итератор конца списка.
void remove(const T &значение);	Удаляет из списка элементы, значения которых равны параметру ***значение.***
template<class UnPred> **void remove_if(UnPred *пред);***	Удаляет из списка значения, для которых истинно значение унарного предиката ***пред.***
reverse_iterator rend(); **const_reverse_iterator rend() const;**	Возвращает обратный итератор начала списка.
void resize (size_type *число,* **T *значение = T());***	Изменяет размер списка в соответствии с параметром ***число.*** Если при этом список удлиняется, то добавляемые в конец списка элементы получают значение, заданное параметром ***значение.***
void reverse ();	Выполняет реверс (т. е. реализует обратный порядок расположения элементов) вызывающего списка.
size_type size() const;	Возвращает хранящееся на данный момент в списке число элементов.
void sort(); **template<class Comp>** **void sort Comp *ф_сравн*);**	Сортирует список. Во второй форме для определения того, является ли значение одного элемента меньшим, чем значение другого, может задаваться функция сравнения ***ф_сравн***.
void splice (iterator i,list<T, Allocator *&объект*);	Вставляет содержимое объекта ***объект*** в вызывающий список. Место вставки определяется итератором ***i***. После выполнения операции ***объект*** становится пустым.
void splice (iterator i, list<T, Allocator> *&объект*, iterator *элемент*);	Удаляет элемент, на который указывает итератор ***элемент***, из списка, хранящегося в объекте ***объект***, и сохраняет его в

	вызывающем списке. Место вставки определяется итератором ***i***.
void splice(iterator i, list<T, Allocator &*объект*, iterator *начало*, iterator *конец*);	Удаляет диапазон элементов, обозначенный итераторами ***начало*** и ***конец***, из списка, хранящегося в объекте объект, и сохраняет его в вызывающем списке. Место вставки определяется итератором ***i.***
void swap(list<T, Allocator> &*объект*);	Обменивает элементы из вызывающего списка с элементами из объекта ***объект***.
void unique(); **template<class BinPred>** **void unique(BinPred *пред*);**	Удаляет из вызывающего списка парные элементы. Во второй форме для выяснения уникальности элементов используется предикат ***пред***.

Примеры

1. Ниже представлен пример простого списка.

```
// Основные операции списка
#include <iostream>
#include <list>
using namespace std;

int main()
{
   list<char> lst; // создание пустого списка
   int i;

   for(i=0; i<10; i++) lst.push_back('A' + i);

   cout << "Размер = " << lst.size() << endl;

   list<char>::iterator p;

   cout << "Содержимое: ";
   while(!lst.empty()) {
      p = lst.begin();
      cout << *p;
      lst.pop_front();
   }

   return 0;
}
```

После выполнения программы на экране появится следующее:

```
Размер =10
Содержимое: ABCDEFGHIJ
```

В этой программе создается список символов. Сначала создается пустой список. Затем туда помещается десять символов (буквы от **A** до **J** включительно). Эта операция выполняется с помощью функции **push_back(),** которая помещает каждое следующее значение в конец существующего списка. Далее размер списка выводится на экран. После этого организуется вывод на экран содержимого списка, для чего каждый раз последовательно извлекают,

выводят на экран и удаляют очередной первый элемент списка. Этот процесс продолжается, пока список не опустеет.

2. В предыдущем примере, пройдя список от начала до конца, мы его опустошили. Это, конечно, не обязательно. Ниже представлена переработанная версия программы.

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

int main()
{
   list<char> lst;
   int i;

   for(i=0; i<10; i++) lst.push_back('A' + i);

   cout << "Размер = " << lst.size() << endl;

   list<char>::iterator p = lst.begin();

   cout << "Содержимое: ";
   while(p != lst.end()) {
      cout << *p;
      p++;
   }

   return 0;
}
```

В данной программе итератор **p** инициализируется таким образом, чтобы он указывал на начало списка. Затем при каждом проходе цикла итератор **p** инкрементируется, что заставляет его указывать на следующий элемент списка. Цикл завершается, когда итератор **p** укажет на конец списка.

3. Поскольку список является двунаправленным, размещать элементы в нём можно как с начала списка так и с его конца. В следующей программе создаётся два списка, причём во втором списке организуется обратный первому порядок расположения элементов

```
// Элементы можно размещать не только начиная с начала списка,
// но также и начиная с его конца
#include <iostream>
#include <list>
using namespace std;

int main()
{
   list<char> lst;
   list<char> revlst;
   int i;

   for(i=0; i<10; i++) lst.push_back('A' + i);

   cout << "Размер прямого списка = " << lst.size() << endl;
   cout << "Содержимое прямого списка: ";

   list<char>::iterator p;
```

```
  // Удаление элементов из первого списка
  // и размещение их в обратном порядке во втором списке
  while(!lst.empty()) {
    p = lst.begin();
    cout << *p;
    lst.pop_front();
    revlst.push_front(*p);
  }
  cout << endl;

  cout << "Размер обратного списка = ";
  cout << revlst.size() << endl;
  cout << "Содержимое обратного списка: ";
  p = revlst.begin();
  while(p != revlst.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

После выполнения программы на экране появится следующее:

```
Размер прямого списка = 10
Содержимое прямого списка: ABCDEFGHIJ

Размер обратного списка = 10
Содержимое обратного списка: JIHGFEDCBA
```

В данном примере реверс списка **lst** достигается следующим образом: элементы поочерёдно извлекаются из начала списка **lst** и размещаются в начале списка **revlst**. Таким образом в списке **revlst** реализуется обратный порядок расположения элементов.

4. Вызвав функцию-член **sort()**, вы можете отсортировать список. В следующей программе создаётся список случайных символов, а затем эти символы сортируются.

```
// Сортировка списка
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
  list<char> lst;
  int i;

  // заполнение списка случайными символами
  for(i=0; i<10; i++) lst.push_back('A' + (rand()%26));

  cout << "Исходное содержимое: ";
  list<char>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p;
    p++;
  }
  cout << endl;

  // сортировка списка
  lst.sort();
```

```
  cout << "Отсортированное содержимое: ";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

После выполнения программы на экране появится следующее:

Исходное содержимое: PHQGHUMEAY

Отсортированное содержимое: AEGHHMPQUY

5. Отсортированный список можно слить с другим. В результате будет получен новый отсортированный список с содержимым, состоящим из содержимого обоих исходных списков. Новый список остаётся в вызывающем списке, а второй список оказывается пустым. Ниже представлен пример слияния двух списков. В первом находятся символы **ACEGI**, а во втором – **BDFHJ**. После слияния мы получим последовательность **ABCDEFGHIJ**.

```
// Слияние двух списков
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> lst1, lst2;
  list<char> revlst;
  int i;

  for(i=0; i<10; i+=2) lst1.push_back('A' + i);
  for(i=1; i<11; i+=2) lst2.push_back('A' + i);

  cout << "Содержимое первого списка: ";
  list<char>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p;
    p++;
  }
  cout << endl;

  cout << "Содержимое второго списка: ";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p;
    p++;
  }
  cout << endl;

  // Слияние двух списков
  lst1.merge(lst2);
  if(lst2.empty())
    cout << "Теперь второй список пуст\n";

  cout << "Содержимое первого списка после слияния:\n";
  p = lst1.begin();
```

```
  while(p != lst1.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

После выполнения программы на экране появится следующее:

```
Содержимое первого списка: ACEGI

Содержимое второго списка: BDFHJ

Теперь второй список пуст
Содержимое первого списка после слияния:
ABCDEFGHIJ
```

6. В следующем примере список используется для хранения объектов типа **Project**. **Project** – это класс, с помощью которого организуется управление программными проектами. Обратите внимание, что для объектов типа **Project** перегружаются операторы <, >, != и ==. Перегрузки этих операторов требует компилятор Microsoft Visual C++ 5. (Именно этот компилятор использовался при отладке примеров данной главы.) Для других компиляторов может потребоваться перегрузить какие-либо дополнительные операторы. В библиотеке стандартных шаблонов с помощью указанных оператор-функций сравниваются объекты, хранящиеся в контейнере. Хотя список не является контейнером с упорядоченным хранением элементов, тем не менее и здесь при поиске, сортировке или слиянии элементы приходиться сравнивать.

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

class Project {
public:
  char name[40];
  int days_to_completion;
  Project() {
    strcpy(name, " ");
    days_to_completion = 0;
  }
  Project(char *n, int d) {
    strcpy(name, n);
    days_to_completion = d;
  }

  void add_days(int i) {
    days_to_completion += i;
  }

  void sub_days(int i) {
    days_to_completion -= i;
  }

  bool completed() { return !days_to_completion; }

  void report() {
    cout << name << ": ";
```

```
    cout << days_to_completion;
    cout << " дней до завершения\n";
  }
};

bool operator<(const Project &a, const Project &b)
{
  return a.days_to_completion < b.days_to_completion;
}

bool operator>(const Project &a, const Project &b)
{
  return a.days_to_completion > b.days_to_completion;
}

bool operator==(const Project &a, const Project &b)
{
  return a.days_to_completion == b.days_to_completion;
}

bool operator!=(const Project &a, const Project &b)
{
  return a.days_to_completion != b.days_to_completion;
}

int main()
{
  list<Project> proj;
  int i;

 proj.push_back(Project("Разработка компилятора", 35));
 proj.push_back(Project("Разработка электронной таблицы", 190));
 proj.push_back(Project("Разработка STL", 1000));

  list<Project>::iterator p = proj.begin();

  // вывод проектов на экран
  while(p != proj.end()) {
    p->report();
    p++;
  }

  // увеличение сроков выполнения первого проекта на 10 дней
  p = proj.begin();
  p->add_days(10);

  // последовательное завершение первого проекта
  do {
    p->sub_days(5);
    p->report();
  } while(!p->completed());

  return 0;
}
```

После выполнения программы на экране появится следующее:

```
Разработка компилятора: 35 дней до завершения
Разработка электронной таблицы: 190 дней до завершения
Разработка STL: 1000 дней до завершения
Разработка компилятора: 40 дней до завершения
Разработка компилятора: 35 дней до завершения
```

Разработка компилятора: 30 дней до завершения
Разработка компилятора: 25 дней до завершения
Разработка компилятора: 20 дней до завершения
Разработка компилятора: 15 дней до завершения
Разработка компилятора: 10 дней до завершения
Разработка компилятора: 5 дней до завершения
Разработка компилятора: 0 дней до завершения

14.5. Ассоциативные списки

Класс **map** поддерживает ассоциативный контейнер, в котором каждому значению соответствует уникальный ключ. По существу, ключ - это просто имя, которое вы присваиваете значению. После того как значение помещено в контейнер, извлечь его оттуда можно с помощью ключа. Таким образом, в самом общем смысле можно сказать, что ассоциативный список представляет собой список пар ключ/значение. Преимущество ассоциативных списков состоит в возможности получения значения по данному ключу. Например, используя ассоциативный список, можно хранить имена телефонных абонентов в качестве ключей, а номера телефонов в качестве значений. Ассоциативные контейнеры в программировании становятся все более и более популярными.

Как уже упоминалось, в ассоциативном списке можно хранить только уникальные ключи. Дублирования ключей не допускается. Для создания ассоциативного списка с неуникальными ключами используется класс-контейнер **multimap**.

Ниже представлена спецификация шаблона для класса **map**:

template<class Key, class T, class Comp = less<Key>,
class Allocator = allocator<T>class map

Здесь **Key** - это данные типа ключ, **T** - тип данных, предназначенных для хранения (в карте), а **Comp** - функция для сравнения двух ключей, которой по умолчанию является стандартная объект-функция **less().** Ключевое слово **Allocator** задает распределитель памяти (которым по умолчанию является **allocator**).

В классе **map** определены следующие; конструкторы:

explicit map(const Comp *&ф_сравн* = Comp(), const Allocator *&а* = Allocator()) ;
map (const map<Key, T, Comp, Allocator>***&объект***) ;

template<class Inlter>map(lnlter *начало*, Inlter *конец*,
const Comp *&ф_сравн* = CompO , const Allocator *&а* = Allocator ()) ;

Первая форма представляет собой конструктор пустого ассоциативного списка. Вторая форма конструктора предназначена для ассоциативного списка из одинаковых элементов, каждый из которых - это ***объект***. Третья форма - это конструктор ассоциативного списка, содержащего диапазон элементов, заданный итераторами ***начало*** и ***конец***. Функция сравнения ***ф_сравн***, если она присутствует, задает порядок сортировки элементов ассоциативного списка.

Как правило, для любого объекта, заданного в качестве ключа, должны быть определены конструктор по умолчанию и несколько операторов сравнения.

Для класса **map** определяются следующие операторы сравнения:

==, <, <=, !=, >, >=

В табл. 14.4 представлены функции - члены класса **map**. В данной таблице тип **key_type** - это тип ключа, а **key_value** - тип пары ключ/значение (тип **pair<Key, T>**).

Таблица 14.4. *Функции - члены класса* ***map***

Функция-член	Описание
iterator begin(); **const_iterator begin() const;**	Возвращает итератор первого элемента ассоциативного списка.
void clear();	Удаляет все элементы ассоциативного списка.
size_type count **(const key_type &*k*) const;**	Возвращает 1 или 0, в зависимости от того, встречается или нет в ассоциативном списке ключ ***k***.
bool empty() const;	Возвращает истину, если вызывающий ассоциативный список пуст, в противном случае возвращает ложь.
iterator end(); **const_iterator end() const;**	Возвращает итератор конца ассоциативного списка.
pair<iterator, iterator> **equal_range(const key_type &*k*);** **pair<const_iterator, const iterator>** **equal_range(const key_type &*k*) const;**	Возвращает пару итераторов, которые указывают на первый и последний элементы ассоциативного списка, содержащего указанный ключ ***k.***
void erase(iterator *i*);	Удаляет элемент, на который указывает итератор ***i***.
void erase(iterator *начало*, iterator *конец*);	Удаляет элементы, заданные между итераторами ***начало*** и ***конец***.
size_type erase (const key_type &*k*);	Удаляет элементы, соответствующие значению ключа ***k***.
iterator find (const key_type &*k*); **const_iterator find (const key_type &*k*) const;**	Возвращает итератор по заданному ключу ***k***. Если ключ не обнаружен, возвращает итератор конца ассоциативного списка.
allocator_type get_allocator() const;	Возвращает распределитель памяти ассоциативного списка.
iterator insert(iterator *i*, **const value_type &*значение*);**	Вставляет параметр ***значение*** на место элемента или после элемента, заданного итератором ***i***. Возвращает итератор этого элемента.
template<class lnIter> void **insert(lnIter *начало*, InIter *конец*);**	Вставляет последовательность элементов, заданную итераторами ***начало*** и ***конец***.
pair<iterator, bool>insert **(const value_type &*значение*);**	Вставляет ***значение*** в вызывающий ассоциативный список. Возвращает итератор вставленного элемента. Элемент вставляется только в случае, если такого в ассоциативном списке еще нет. При удачной вставке элемента функция возвращает значение **pair<iterator, true>**, в противном случае - **pair<iterator, false>.**
key_compare key_comp() const;	Возвращает объект-функцию сравнения

	ключей.
iterator lower_bound (const key_type &*k*); **const_iterator lower_bound** **(const key_type &*k*) const;**	Возвращает итератор первого элемента ассоциативного списка, ключ которого равен или больше заданного ключа ***k***.
size_type max_size() const;	Возвращает максимальное число элементов, которое можно хранить в ассоциативном списке.
reference operator[] (const key_type &*i*);	Возвращает ссылку на элемент, соответствующий ключу ***i***. Если такого элемента не существует, он вставляется в ассоциативный список.
reverse_iterator rbegin(); **const_reverse_iterator rbegin() const;**	Возвращает обратный итератор конца ассоциативного списка.
reverse_iterator rend(); **const_reverse_iterator rend() const;**	Возвращает обратный итератор начала ассоциативного списка.
size_type size() const;	Возвращает хранящееся на данный момент в ассоциативном списке число элементов.
void swap(map<Key, T, Comp, Allocator> &*обьект*);	Обменивает элементы из вызывающего ассоциативного списка с элементами из объекта ***объект***.
iterator upper_bound (const key_type &*k*); **const_iterator upper_bound** **(const key_type &*k*) const;**	Возвращает итератор первого элемента ассоциативного списка, ключ которого больше заданного ключа ***k***.
value_compare value_comp() const;	Возвращает объект-функцию сравнения значений.

В ассоциативном списке хранятся пары ключ/значение в виде объектов типа **pair**. Шаблон объекта типа **pair** имеет следующую спецификацию:

template<class *Ktype*, class *Vtype*> struct pair {
typedef *Ktype первый_тип*; // тип ключа
typedef *Vtype второй_тип*; // тип значения
***Ktype первый*; II содержит ключ**
***Vtype второй*; II содержит значение**

// конструкторы
pair ();
pair(const *Ktype &k*, const *Vtype &v*) ;
template<class A, class B> pair(const<A, B> &*объект*)

Ранее уже говорилось, что значение переменной ***первый*** содержит ключ и значение, а значение переменной ***второй*** - значение, соответствующее этому ключу.

Создавать пары ключ/значение можно не только с помощью конструкторов класса **pair**, но и с помощью функции **make_pair()**, которая создает объекты типа **pair**, используя типы данных в качестве параметров. Функция **make_pair()** - это родовая функция со следующим прототипом:

template<class *Ktype*, class *Vtype*>
pair<*Ktype*, *Vtype*>make_pair (const *Ktype* &*k*, const *Vtype* &*v*) ;

Как видите, функция возвращает объект типа **pair**, содержащий заданные в качестве параметров функции значения типов ***Ktype*** и ***Vtype***. Преимущество использования функции **make_pair()** состоит в том, что она дает ,возможность компилятору автоматически распознавать типы предназначенных для хранения объектов, и вам не нужно указывать их явно.

Примеры

1. В следующей программе на примере ассоциативного списка, предназначенного для хранения десяти пар ключ/значение, иллюстрируются основы использования ассоциативных списков. Ключом здесь является символ, а значением - целое. Пары ключ/значение хранятся следующим образом:

A	**0**
B	**1**
C	**2**
. . .	
J	**9**

Поскольку пары хранятся именно таким образом, то, когда пользователь набирает на клавиатуре ключ (т. е. одну из букв от **A** до **J**), программа выводит на экран соответствующее этому ключу значение.

```
// Иллюстрация возможностей ассоциативного списка
#include <iostream>
#include <map>
using namespace std;

int main()
{
   map<char, int> m;
   int i;

   // размещение пар в ассоциативном списке
   for(i=0; i<10; i++) {
      m.insert(pair<char, int>('A' + i, i));
   }

   char ch;
   cout << "Введите ключ: ";
   cin >> ch;

   map<char, int>::iterator p;

   // поиск значения по заданному ключу
   p = m.find(ch);
   if(p != m.end())
      cout << p->second;
   else
      cout << "Такого ключа в ассоциативном списке нет\n";

   return 0;
}
```

Обратите внимание на использование класса-шаблона **pair** для образования пар ключ/значение. Типы данных, указанные в классе-шаблоне **pair**, должны соответствовать типам данных, хранящимся в ассоциативном списке.

После того как ассоциативный список инициализирован парами ключ/значение, найти нужное значение по заданному ключу можно с помощью функции **find()**. Функция **find()** возвращает итератор соответствующего ключу элемента или итератор конца ассоциативного списка, если указанный ключ не найден. Когда соответствующее ключу значение найдено, оно сохраняется в качестве второго члена класса-шаблона **pair**.

2. В предыдущем примере типы пар ключ/значение были указаны явно в конструкции **pair <char, int>**. Хотя такой подход совершенно правилен, часто проще использовать функцию **make_pair()**, которая создаёт пары объектов на основе типов данных своих параметров.

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

int main()
{
   map<char, int> m;
   int i;

   // размещение пар в ассоциативном списке
   for(i=0; i<10; i++) {
      m.insert(make_pair(char)('A' + i, i));
   }

   char ch;
   cout << "Введите ключ: ";
   cin >> ch;

   map<char, int>::iterator p;

   // поиск значения по заданному ключу
   p = m.find(ch);
   if(p != m.end())
      cout << p->second;
   else
      cout << "Такого ключа в ассоциативном списке нет\n";

   return 0;
}
```

Данный пример отличается от предыдущего только строкой

```
m.insert (make_pair(char) ('A' + i, i));
```

В данном случае, чтобы в операции сложения **'A' + i**, чтобы не допустить автоматического преобразования в тип **int**, используется приведение к типу **char**. Во всём остальном процесс определения типов объектов выполняется автоматически.

4. Так же как и в других контейнерах, в ассоциативных списках можно хранить создаваемые вами типы данных. Например, в представленной ниже программе создаётся ассоциативный список с соответствующими словам антонимами. С этой целью используются два класса **word** (слово) и **opposite** антоним). Поскольку для

ассоциативных списков поддерживается отсортированный список ключей, в программе для объектов типа **word** определяется оператор <. Как правило, оператор < необходимо перегружать для всех классов, объекты которых предполагается использовать в качестве ключей. (Помимо оператора < в некоторых компиляторах может потребоваться определить дополнительные операторы сравнения.)

```
// Ассоциативный список слов и антонимов
#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class word {
   char str[20];
public:
   word() { strcpy(str, ""); }
   word(char *s) { strcpy(str, s); }
   char *get() { return str; }
};

// для объектов типа word следует определить оператор < (меньше)
bool operator<(word a, word b)
{
   return strcmp(a.get(), b.get()) < 0;
}

class opposite {
   char str[20];
public:
   opposite() { strcmp(str, ""); }
   opposite(char *s) { strcpy(str, s); }
   char *get() { return str; }
};

int main()
{
   map<word, opposite> m;

   // размещение в ассоциативном списке слов и антонимов
   m.insert(pair<word, opposite>
      (word("да"),opposite("нет")));
   m.insert(pair<word, opposite>
      (word("хорошо"),opposite("плохо")));
   m.insert(pair<word, opposite>
      (word("влево"),opposite("вправо")));
   m.insert(pair<word, opposite>
      (word("вверх"),opposite("вниз")));

   // поиск антонима по заданному слову
   char str[80];
   cout << "Введите слово: ";
   cin >> str;

   map<word, opposite>::iterator p;

   p = m.find(word(str));
   if(p != m.end())
      cout << "Антоним: " << p->second.get();
   else
      cout << "Такого слова в ассоциативном списке нет\n";
   cout << "ob1: " << ob1.geta() << endl;
```

```
  cout << "ob2: " << ob2.geta() << endl;

  return 0;
}
```

В данном примере любой объект, который вводится в ассоциативный список, представляет собой символьный массив для хранения заканчивающейся нулем строки. Далее в этой главе будет показано, как упростить эту программу, используя стандартный тип данных **string**.

14.6. Алгоритмы

Как уже объяснялось, алгоритмы предназначены для разнообразной обработки контейнеров. Хотя, в каждом контейнере поддерживается собственный базовый набор операций, стандартные алгоритмы обеспечивают более широкие и комплексные действия. Кроме этого, они позволяют одновременно работать с двумя контейнерами разных типов. Для доступа к алгоритмам библиотеки стандартных шаблонов в программу необходимо включить заголовок **<algorithm>**.

В библиотеке стандартных, шаблонов определяется большое число алгоритмов, которые систематизированы в табл. 14.5. Все алгоритмы представляют собой функции-шаблоны. Это означает, что их можно использовать с контейнерами любых типов. Наиболее показательные варианты такого использования приведены в примерах данного раздела.

Таблица 14.5. *Алгоритмы библиотеки стандартных шаблонов*

Алгоритм	Назначение
adjacent_find	Выполняет поиск смежных парных элементов в последовательности. Возвращает итератор первой пары.
binary_search	Выполняет бинарный поиск в упорядоченной последовательности.
copy	Копирует последовательность.
copy_backward	Аналогична функции **copy()**, за исключением того, что перемещает в начало последовательности элементы из ее конца.
count	Возвращает число элементов в последовательности.
count_if	Возвращает число элементов в последовательности, удовлетворяющих некоторому предикату.
equal	Определяет идентичность двух диапазонов.
equal_range	Возвращает диапазон, в который можно вставить элемент, не нарушив при этом порядок следования элементов в последовательности.
fill **fill_n**	Заполняет диапазон заданным значением.
find	Заполняет диапазон заданным значением. Выполняет поиск диапазона для значения и возвращает первый найденный элемент.
find_end	Выполняет поиск диапазона для

	подпоследовательности. Функция возвращает итератор конца подпоследовательности внутри диапазона.
find_first_of	Находит первый элемент внутри последовательности, парный элементу внутри диапазона.
find if	Выполняет поиск диапазона для элемента, для которого определенный пользователем унарный предикат возвращает истину.
for_each	Назначает функцию диапазону элементов.
generate **generate_n**	Присваивает элементам в диапазоне значения, возвращаемые порождающей функцией.
includes	Определяет, включает ли одна последовательность все элементы другой последовательности.
inplace_merge	Выполняет слияние одного диапазона с другим. Оба диапазона должны быть отсортированы в порядке возрастания элементов. Результирующая последовательность сортируется.
iter_swap	Меняет местами значения, на которые указывают два итератора, являющиеся аргументами функции.
lexicographlcal_compare	Сравнивает две последовательности в алфавитном порядке.
lower_bound	Обнаруживает первое значение в последовательности, которое не меньше заданного значения.
make_heap	Выполняет пирамидальную сортировку последовательности (пирамида, на английском языке heap, - полное двоичное дерево, обладающее тем свойством, что значение каждого узла не меньше значения любого из его дочерних узлов. - Примеч. пер.).
max	Возвращает максимальное из двух значений.
max_element	Возвращает итератор максимального элемента внутри диапазона.
merge	Выполняет слияние двух упорядоченных последовательностей, а результат размещает в третьей последовательности.
in	Возвращает минимальное из двух значений.
min_element	Возвращает итератор минимального элемента внутри диапазона.
mismatch	Обнаруживает первое несовпадение между элементами в двух последовательностях. Возвращает итераторы обоих несовпадающих элементов.
next_permutation	Образует следующую перестановку (permutation) последовательности.
nth element	Упорядочивает последовательность таким

	образом, чтобы все элементы, меньшие заданного элемента ***E***, располагались перед ним, а все элементы, большие заданного элемента ***E***, - после него.
partial_sort	Сортирует диапазон.
partial_sort_copy	Сортирует диапазон, а затем копирует столько элементов, сколько войдет в результирующую последовательность.
partition	Упорядочивает последовательность таким образом, чтобы все элементы, для которых предикат возвращает истину, располагались перед элементами, для которых предикат возвращает ложь.
pop_heap	Меняет местами первый и предыдущий перед последним элементы, а затем восстанавливает пирамиду.
prev_permutation	Образует предыдущую перестановку последовательности.
push_heap	Размещает элемент на конце пирамиды.
random_shuffle	Беспорядочно перемешивает последовательность.
remove **remove_if** **remove_copy** **remove_copy_if**	Удаляет элементы из заданного диапазона.
replace **replace_if** **replace_copy** **replace_copy_if**	Заменяет элементы внутри диапазона.
reverse **reverse_copy**	Меняет порядок сортировки элементов диапазона на обратный.
rotate	Выполняет циклический сдвиг влево элементов в диапазоне.
rotate_copy	Выполняет поиск подпоследовательности внутри последовательности.
search	Выполняет поиск последовательности заданного числа одинаковых элементов.
search_n	Создает последовательность, которая содержит различающиеся участки двух упорядоченных наборов.
set_difference	Создает последовательность, которая содержит одинаковые участки двух упорядоченных наборов.
set_intersection	Создает последовательность, которая содержит симметричные различающиеся участки двух упорядоченных наборов.
set_symmetric_difference	Создает последовательность, которая содержит симметричные различающиеся участки двух упорядоченных наборов.
set_union	Создает последовательность, которая содержит объединение (union) двух упорядоченных наборов.

sort	Сортирует диапазон.
sort_heap	Сортирует пирамиду внутри диапазона.
stable_partition	Упорядочивает последовательность таким образом, чтобы все элементы, для которых предикат возвращает истину, располагались перед элементами, для которых предикат возвращает ложь. Разбиение на разделы остается постоянным; относительный порядок расположения элементов последовательности не меняется.
stable_sort	Сортирует диапазон. Одинаковые элементы не переставляются.
swap_ranges	Меняет местами два значения.
transform	Меняет местами элементы в диапазоне.
unique	Назначает функцию диапазону элементов и сохраняет результат в новой последовательности.
unique_copy	Удаляет повторяющиеся элементы из диапазона.
upper_bound	Обнаруживает последнее значение в последовательности, которое не больше некоторого значения.

Примеры

1. Одними из самых простых алгоритмов являются алгоритмы **count()** и **count_if().** Ниже представлены их основные формы:

template<class InIter, class T>
size_t count(InIter *начало*, InIter *окончание*, const T &*значение*) ;

template<class InIter, class T>
size_t count(InIter *начало*,
InIter *окончание*, UnPred &*ф_предикат*);

Алгоритм **count()** возвращает число элементов в последовательности, начиная с элемента, обозначенного итератором ***начало***, и заканчивая элементом, обозначенным итератором ***окончание***, значение которых равно параметру ***значение***. Алгоритм **count_if()** возвращает число элементов в последовательности, начиная с элемента ***начало*** и заканчивая элементом ***окончание,*** для которых унарный предикат ***ф_предикат*** возвращает истину.

В следующей программе демонстрируются алгоритмы **count()** и **count_if().**

```
// Демонстрация алгоритмов count и count_if
#include <iostream>
#include <vector>
#include <algoritm>
using namespace std;

/* это унарный предикат, который определяет, является ли значение четным
*/
```

```
bool even(int x)
{
   return !(x%2);
}

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

   for(i=0; i<20; i++) {
      if(i%2) v.push_back(1);
      else v.push_back(2);
   }

   cout << "Последовательность: ";
   for(i=0; i<v.size(); i++) cout << v[i] << " ";
   cout << endl;

   int n;
   n = count(v.begin(), v.end(), 1);
   cout << n << " элементов равно 1\n";

   n = count_if(v.begin(), v.end(), even);
   cout << n << " четных элементов\n";

   return 0;
}
```

После выполнения программы на экране появится следующее:

```
Последовательность: 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
10 элементов равно 1
10 чётных элементов
```

Программа начинается с создания 20-элементного вектора, содержащего чередующиеся значения 1 и 2. Для подсчёта единиц используется алгоритм **count()**, а для подсчёта чётных элементов – алгоритм **count_if().** Отметьте, как программируется унарный предикат **even()**. Все унарные предикаты получают в качестве параметра объект, тип которого тот же, что и тип объектов контейнера, для работы с которым предназначен предикат. В зависимости от значения этого объекта унарный предикат должен возвращать истину либо ложь.

2. Иногда полезно генерировать новую последовательность, состоящую только из определённых фрагментов исходной последовательности. Одним из предназначенных для этого алгоритмов является алгоритм **remove_copy()** основная форма которого представлена ниже:

template<class InIter, class OutIter, class T>
OutIter remove_copy (InIter *начало*, InIter *окончание*, OutIter, *результат*, const T *&значение*);

Алгоритм **remove_copy()** копирует элементы, равные параметру ***значение***, из заданного итераторами ***начало*** и ***окончание*** диапазона и размещает результат в последовательности, обозначенный итератором ***результат***. Алгоритм возвращает итератор конца новой последовательности. Результирующий контейнер должен быть достаточно велик для хранения новой последовательности.

В следующем примере показана работа алгоритма **remove_copy()**. Сначала в программе создаётся чередующаяся последовательность значений 1 и. Затем из последовательности удаляются все единицы.

```
// Демонстрация алгоритма remove_copy
#include <iostream>
#include <vector>
#include <algoritm>
using namespace std;

int main()
{
   vector<int> v, v2(20);
   int i;

   for(i=0; i<20; i++) {
      if(i%2) v.push_back(1);
      else v.push_back(2);
   }

   cout << "Последовательность: ";
   for(i=0; i<v.size(); i++) cout << v[i] << " ";
   cout << endl;

   // удаление единиц
   remove_copy(v.begin(), v.end(), v2.begin(), 1);
   cout << "Результат: ";
   for(i=0; i<v2.size(); i++) cout << v2[i] << " ";
   cout << endl;

   return 0;
}
```

После выполнения программы на экране появится следующее:

```
Последовательность: 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
Результат: 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0
```

3. Ещё одним полезным алгоритмом является алгоритм **reverse()**, который меняет порядок расположения элементов последовательности на обратный. Ниже представлена основная форма этого алгоритма:

template<class Bilter>
void reverse(Bilter *начало,* Bilter *окончание*);

Алгоритм **reverse()** меняет братный порядок расположения элементов в диапазоне, заданном итераторами ***начало*** и ***окончание.***

В следующем примере показана работа алгоритма **reverse()**.

```
// Демонстрация алгоритма reverse
#include <iostream>
#include <vector>
#include <algoritm>
using namespace std;

int main()
```

```
{
   vector<int> v;
   int i;

   for(i=0; i<10; i++) v.push_back(i);

   cout << "Исходная последовательность: ";
   for(i=0; i<v.size(); i++) cout << v[i] << " ";
   cout << endl;

   reverse(v.begin(), v.end());

   cout << "Обратная последовательность: ";
   for(i=0; i<v.size(); i++) cout << v[i] << " ";

   return 0;
}
```

После выполнения программы на экране появится следующее:

```
Исходная последовательность: 0 1 2 3 4 5 6 7 8 9
Обратная последовательность: 9 8 7 6 5 4 3 2 1 0
```

5. Одним из наиболее интересных алгоритмов является алгоритм **transform()**, который модифицирует каждый элемент некоторого диапазона в соответствии с заданной вами функцией. Алгоритм **transform()** имеет две основне формы:

template<class Inlter, class Outlter, class Func>
Outlter transform(Inlter *начало*, Inlter *окончание*, Outlter *результат*, Func *унарная_функция*);

template<class Inlter1, class Inlter2, class Outlter, class Func>
Outlter transform(Inlter *начало1*, Inlter *окончание1*, Inlter *начало2*, Outlter *результат*, Func *бинарная_функция*);

Алгоритм **transform()** применяет функцию к диапазону элементов и сохраняет результата в месте, определённом итератором ***результат.*** В первой форме диапазон задаётся итераторами ***начало*** и ***окончание***, а применяемой функцией является ***унарная_функция***. Эта функция в качества параметра получает значение элемента и должна возвратить модифицированный элемент. Во второй форме модификация осуществляется с помощью бинарной оператор-функции ***бинарная_функция***, которая в качестве первого параметра получает значение элемента из предназначенной для модификации последовательности, а в качестве второго параметра – элемент из второй последовательности. Обе версии возвращают итератор конца итоговой последовательности.

В следующей программе для модификации используется функция **xform()**, которая возводит в квадрат элементы списка. Обратите внимание, что итоговая последовательность хранится в том же списке, что и исходная последовательность.

```
// Пример использования алгоритма transform
#include <iostream>
#include <list>
#include <algoritm>
using namespace std;

// Простая функция модификации
```

```
int xform(int i) {
   return i * i; // квадрат исходного значения
}

int main()
{
   list<int> x1;
   int i;

   // размещение значений в списке
   for(i=0; i<10; i++) v.push_back(i);

   cout << "Исходное содержимое списка x1: ";
   list<int>::iterator p = x1.begin();
   while(p != x1.end()) {
      cout << *p << " ";
      p++;
   }

   cout << endl;

   // модификация элементов списка x1
   p = transform(x1.begin(), x1.end(), x1.begin(), xform);

   cout << "Модифицированное содержимое списка x1: ";
   p = x1.begin();
   while(p != x1.end()) {
      cout << *p << " ";
      p++;
   }

   return 0;
}
```

После выполнения программы на экране появится следующее:

```
Исходное содержимое списка x1: 0 1 2 3 4 5 6 7 8 9
Модифицированное содержимое списка x1: 0 1 4 9 16 25 36 49 64 81
```

Как видите, возведён в квадрат каждый элемент списка **x1**.

14.7. Строковый класс

Как известно, в C++ встроенный строковый тип данных сам по себе не поддерживается. Тем не менее для обработки строк здесь имеется две возможности. Во-первых, можно использовать хорошо вам знакомый оканчивающийся нулем символьный массив. О таком массиве иногда говорят как о строке в стиле C (C string). Второй метод, который и рассматривается в этом разделе, подразумевает использование объектов типа **string**.

Фактически, класс **string** является конкретизацией более общего класса-шаблона **basic_string**. На самом деле у класса **basic_string** имеется два производных класса: класс string, который поддерживает строки 8-разрядных символов, и **wstring**, который поддерживает строки широких символов. Поскольку при обычном программировании чаще всего имеют дело именно с 8-разрядными символами, мы рассмотрим только версию **string** базового класса **basic_string**.

Перед тем как начать изучение класса **string**, важно понять, почему он включен в библиотеку классов C++. Стандартные классы появились в C++ не случайно. Фактически, включению в библиотеку каждого нового класса сопутствовало множество споров и дискуссий. Добавление в C++ класса string на первый взгляд кажется исключением из этого правила, поскольку в C++ в качестве строк уже поддерживаются оканчивающиеся нулем массивы. Тем не менее это далеко не так, и вот почему: оканчивающиеся нулем символьные массивы нельзя обрабатывать посредством стандартных операторов C++ и они не могут быть частью обычных выражений C++. Например, рассмотрим следующий фрагмент программы:

```
char sl[80], s2[80], s3[80];

sl = "раз"; //не допускается
s2 = "два"; // не допускается
s3 = sl + s2; // ошибка, снова не допускается
```

Как показано в комментариях, в C++ нельзя использовать оператор присваивания, чтобы дать символьному массиву новое значение (за исключением инициализации), а для конкатенации двух строк нельзя использовать оператор сложения. Эти операции приходится выполнять с помощью показанных ниже библиотечных функций:

```
strcpy(sl, "раз");
strcpy(s2, "два");
strcpy(s3, sl);
strcpy(s3, s2);
```

Поскольку оканчивающиеся нулем символьные массивы по своей сути технически не являются типами данных, к ним нельзя применять операторы C++. Это приводит к тому, что даже самые элементарные операции со строками становятся чрезвычайно запутанными. Невозможность использования стандартных операторов C++ для работы с оканчивающимися нулем символьными массивами и стала основной причиной разработки стандартного строкового класса. Вспомните, когда вы в C++ определяете класс, вы определяете новый тип данных, который может быть полностью интегрирован в среду программирования C++. Само собой это означает, что относительно нового класса можно перегружать операторы. Таким образом, благодаря добавлению в C++ стандартного класса **string**, становится возможным обрабатывать строки точно таким же образом, каким обрабатываются данные других типов, а именно с помощью операторов.

Имеется, однако, и еще один довод в пользу использования стандартного класса **string** - это обеспечение безопасности. Неопытный или неосторожный программист может очень легко выйти за границы массива, в котором хранится оканчивающаяся нулем строка. Например, рассмотрим стандартную функцию копирования строк **strcpy().** В этой функции совершенно отсутствуют какие бы то ни было атрибуты, предназначенные для контроля границ целевого массива. Если в исходном массиве оказывается больше символов, чем может поместиться в целевом массиве, вполне возможна (и даже весьма вероятна) программная или даже системная ошибка. Как вы в дальнейшем увидите, стандартный класс string предотвращает саму возможность возникновения подобных ошибок.

Итак, для включения в C++ стандартного класса **string** имеется три довода: совместимость (теперь строка становится типом данных), удобство (можно использовать стандартные операторы C++) и безопасность (границы массива не нарушаются). Запомните, что это не доводы в пользу отказа от обычных, оканчивающихся нулем массивов. Они остаются наиболее эффективным способом реализации символьных строк. Тем не менее, если при создании приложения скорость выполнения программы не является доминирующим

фактором, новый класс **string** предоставляет вам безопасный и полностью интегрированный в среду программирования C++ способ обработки строк.

Хотя строковый класс традиционно не считают частью библиотеки стандартных шаблонов, тем не менее, это один из определенных в C++ классов-контейнеров. Это, в частности, означает, что он поддерживает описанные в предыдущем разделе алгоритмы. Кроме этого, для обработки строк имеются дополнительные возможности. Чтобы получить доступ к классу **string**, в программу следует включить заголовок **<string>**.

Класс string очень велик, в нем имеется множество конструкторов и функций-членов. Помимо этого многие функции-члены имеют массу перегруженных форм. По этой причине в одной главе невозможно рассказать о всех членах класса, **string**. Вместо этого мы исследуем только некоторые, основные его возможности. После получения базовых знаний о работе класса **string** в целом, все остальное вы сможете легко понять самостоятельно.

В классе **string** поддерживается несколько конструкторов. Ниже представлены прототипы трех из них, которые чаще всего используются:

string() ;
string(const char * строка);
string(const string &строка);

В первой форме создается пустой объект типа **string**. Во второй форме - объект типа **string** из оканчивающейся нулем строки, обозначенной указателем ***строка***. Эта форма обеспечивает преобразование из оканчивающейся нулем строки в объект типа **string**. В третьей форме объект типа string создается из другого объекта типа **string**.

Ниже перечислена часть операторов, допустимых при работе с объектами типа **string**:

Оператор	Значение
=	Присваивание
+	Конкатенация
+=	Присваивание с конкатенацией
==	Равенство
!=	Неравенство
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
[]	Индекс
<<	Вывод
>>	Ввод

Указанные операторы позволяют использовать объекты типа **string** в обычных выражениях и отказаться от вызовов специализированных функций, например, функций **strcpy()** или **strcat()**. Как правило, объекты типа string в выражениях можно записывать вместе с обычными, оканчивающимися нулем строками. Например, объект типа **string** можно присвоить оканчивающейся нулем строке.

Оператор + можно использовать для конкатенации объекта типа **string** с другим объектом типа **string** или для конкатенации объекта типа **string** со строкой в стиле C. Поддерживаются следующие варианты:

string + string
string + C-string
C-string + string

Кроме этого, оператор + можно использовать для присоединения одиночного символа к концу строки.

В классе **string** определена константа **npos**, обычно равная -1. Эта константа отражает максимально возможную длину строки.

Несмотря на то, что большая часть операций над строками может быть выполнена с помощью строковых операторов, для некоторых наиболее сложных или необычных операций нужны функции - члены класса **string**. Хотя их слишком много для одной главы, о некоторых наиболее полезных здесь будет рассказано. Чтобы присвоить одну строку другой используется функция **assign().** Ниже представлены две основные формы этой функции:

string fiassign(const string *&объект_строка*, size type *начало*, size_type *число*);
string &assign(const char * строка, size_type *число*);

В первой форме несколько символов, количество которых равно параметру ***число*** из объекта ***объект_строка***, начиная с индекса начало, присваиваются вызывающему объекту. Во второй форме вызывающему объекту присваивается первые несколько символов, количество которых равно параметру ***число*** из оканчивающейся нулем строки ***строка***. В обоих случаях функция возвращает ссылку на вызывающий объект. Очевидно, что для присваивания одной целой строки другой гораздо проще использовать оператор =. Функция ***assign()*** может понадобиться только при присваивании части строки.

Присоединить часть одной строки к другой можно с помощью функции-члена **append()**. Ниже представлены две основные формы этой функции:

string &append(const string &*обьект_строка*, size type *начало*, size_type *число*);
string &append(const char * *строка*, size_type *число*);

В первой форме несколько символов, количество которых равно параметру число из объекта ***объект_строка***, начиная с индекса ***начало*** присоединяются к вызывающему объекту. Во второй форме к вызывающему объекту присоединяются первые несколько символов, количество которых равно параметру число, из оканчивающейся нулем строки ***строка***. В обоих случаях функция возвращает ссылку на вызывающий объект. Очевидно, что для присоединения одной целой строки к другой гораздо проще использовать оператор +. Функция **append()** может понадобиться только при присоединении части строки.

С помощью функций **insert()** и **rep1ace()** можно соответственно вставлять или заменять символы в строке. Ниже представлены прототипы основных форм этих функций:

string &insert(size_type *начало*, const string &*объект_строка*) ;
string &insert(size_type *начало*, const string &*объект строка*, size_type *начало вставки*, size type *число*);
string &replace(size_type *начало*, size_type *число*, const string &*объект_строка*) ;
string fireplace(size_type *начало*, size_type *исх_номер*, const string &*объект строка*, size_type *начало_замены*, size_type *число_замены*) ;

В первой форме функции **insert()** объект ***объект_строка*** вставляется в вызывающую строку по индексу ***начало***. Во второй форме функции **insert()** число символов из объекта ***объект_строка***, начиная с индекса ***начало_вставки***, вставляется в вызывающую строку по индексу ***начало***.

В первой форме функции **replace() число** символов, начиная с индекса начало, заменяется в вызывающей строке объектом ***объект_строка***. Во второй форме функции **replace()** в вызывающей строке заменяется ***исх_число*** символов, начиная с индекса ***начало***, при этом из объекта ***объект_строка*** берутся ***число_замены*** символов, начиная с индекса ***начало_замены***. В обоих случаях функция возвращает ссылку на вызывающий объект.

Удалить символы из строки можно с помощью функции **erase().** Ниже показана одна из форм этой функции:

string &erase(size type *начало* == 0, size type *число* = npos);

Функция удаляет ***число*** символов из вызывающей строки, начиная с индекса ***начало***. Возвращаемым значением является ссылка на вызывающий объект.

В классе **string** поддерживается несколько функций-членов, предназначенных для поиска строк. Среди них имеются функции **find()** и **rfind()**. Ниже показаны прототипы основных версий этих функций:

size type find(const string &*объект_строка*, size type *начало* = 0) const;
size_type rfind(const string &*объект_строка*, size type *начало* = npos) const;

Начиная с индекса ***начало*** функция **find()** ищет в вызывающей строке первое совпадение со строкой, содержащейся в объекте ***объект_строка***. Если искомая строка найдена, функция **find()** возвращает индекс вызывающей строки, соответствующий найденному совпадению. Если искомая строка не найдена, функция **find()** возвращает значение **npos**. В противоположность функции **find()**, функция **rfind()**, начиная с индекса ***начало***, но в обратном направлении, ищет в вызывающей строке первое совпадение со строкой, содержащейся в объекте ***объект_строка***. (То есть ищет последнее совпадение со строкой, содержащейся в объекте ***объект_строка***.) Если искомая строка найдена, функция **rfind()** возвращает индекс вызывающей строки, соответствующий найденному совпадению. Если искомая строка не найдена, функция **rfind()** возвращает значение **npos**.

Для сравнения целых строковых объектов удобнее всего пользоваться описанными ранее, перегруженными операторами отношения. Тем не менее, если вы захотите сравнить части строк, вам понадобится функция-член **compare()**. Ниже представлен прототип этой функции:

int compare(size_type *начало*, size_type *число*, const string &*объект_строка*) const;

Здесь с вызывающей строкой сравниваются число символов объекта ***объект_строка***, начиная с индекса ***начало***. Если вызывающая строка меньше, чем ***объект_строка***, функция **compare()** возвращает отрицательное значение. Если вызывающая строка больше, чем ***объект_строка***, функция **compare()** возвращает положительное значение. Если вызывающая строка равна объекту ***объект_строка***, возвращаемое значение функции **compare()** равно нулю.

Хотя объекты типа **string** сами по себе очень удобны, иногда у вас все же будет возникать необходимость в версии строки в виде массива символов, оканчивающихся нулем. Например, объект типа **string** можно использовать для образования имени файла. Однако при открытии файла вам придется задавать указатель на стандартную, оканчивающуюся нулем строку. Для решения проблемы в классе **string** имеется функция-член **c_str(),** прототип которой показан ниже:

const char *c_str() const;

Функция возвращает указатель на оканчивающуюся нулем версию строки, содержащуюся в вызывающем объекте типа **string**. Оканчивающаяся нулем строка не должна меняться. Кроме этого, если над объектом типа **string** выполнялись какие-либо другие операции; правильность выполнения функции **c_str()** не гарантируется.

Поскольку класс string является контейнером, в нем поддерживаются функции **begin()** и **end(),** возвращающие соответственно итератор начала и конца строки. Также поддерживается функция **size(),** возвращающая текущее число символов строки.

Примеры

Хотя мы уже привыкли к традиционным строкам в стиле С, в С++ класс **string** делает обработку строк существенно проще. Например, при работе с объектами типа **string** для присваивания строк можно использовать оператор =, для конкатенации строк – оператор +, а для сравнения строк – различные операторы сравнения. В следующей программе показаны эти операции.

```
// Короткий пример использования строкового класса
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string str1("Представление строк");
   string str2("Вторая строка");
   string str3;

   // присваивание строк
   str3 = str1;
   cout << str1 << "\n" << str3 << "\n";

   // конкатенация двух строк
   str3 = str1 + str2;
   cout << str3 << "\n";

   // сравнение строк
   if(str3 > str1) cout << "str3 > str1\n";
   if(str3 == str1+str2) cout << "str3 == str1+str2\n";

   // строковому объекту можно присвоить обычную строку
   str1 = "Это обычная строка\n";
   cout << str1;

   // создание строкового объекта
   // с помощью другого строкового объекта
```

```
   string str4(str1);
   cout << str4;

   // ввод строки
   cout << "Введите строку: ";
   cin >> str4;
   cout << str4;

   return 0;
}
```

После выполнения программы на экране появится следующее:

```
Представление строк
Представление строк
Представление строк Вторая строка
str3 > str1
str3 == str1+str2
Это обычная строка
Это обычная строка
Введите строку: Привет
Привет
```

Как видите, с объектами типа **string** можно обращаться так же, как и со встроенными типами данных C++. Это, фактически, и есть главное достоинство строкового класса.

Отметьте простоту манипулирования со строками: для конкатенации строк используется обычный оператор +, а для их сравнения – обычный оператор >. Чтобы выполнить те же операции для оканчивающихся нулём строк в стиле C, вам пришлось бы вызывать функции **strcat()** и **strcmp()**, что, согласитесь, гораздо менее удобно. Поскольку объекты типа **string** можно совершенно свободно указывать в выражениях вместе с оканчивающимися нулём строками в стиле C, то никаких неприятностей от их использования быть не может, а выгоды, наоборот, - очевидны.

Имеется ещё одна деталь, на которую следует обратить внимание в предыдущей программе: размеры строк не задаются. Объекты типа **string** автоматически настраиваются На хранение строк требуемой длины. Таким образом, когда вы выполняете присваивание или конкатенацию строк, размер целевой строки автоматически вырастает ровно настолько, насколько это нужно для размещения новой строки. Этот динамический аспект использования объектов типа **string** следует всегда принимать во внимание при выборе варианта представления строк в ваших программах. (Как уже отмечалось, стандартные оканчивающиеся нулём строки *являются* возможным источником нарушения границ массивов).

2. В следующей программе демонстрируются функции **insert(), erase()** и **replace().**

```
// Использование функций insert(), erase() и replace()
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string str1("Это проверка");
   string str2("АБВГДЕЖ");
```

```
    cout << "Исходные строки:\n"
    cout << "str1: " << str1 << endl;
    cout << "str2: " << str2 << "\n\n";

    // работа функции insert()
    cout << "Вставка строки str2 в строку str1:\n"
    str1.insert(4, str2);
    cout << str1 << "\n\n";

    // работа функции erase()
    cout << "Удаление семи символов из строки str1:\n"
    str1.erase(4, 7);
    cout << str1 << "\n\n";

    // работа функции replace()
    cout << "Замена восьми символов из str1 символами из str2:\n"
    str1.replace(4, 8, str2);
    cout << str1 << "\n\n";

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Исходные строки:
str1:  Это проверка
str2: АБВГДЕЖ

Вставка строки str2 в строку str1:
Это АБВГДЕЖпроверка

Удаление семи символов из строки  str1:
Это проверка

Замена восьми символов из str1 символами из str2:
Это АБВГДЕЖ
```

3. Поскольку класс **string** определяет тип данных, появляется возможность создавать контейнеры для хранения объектов типа **string**. Например, ниже представлена усовершенствованная версия программы создания ассоциативного списка для хранения слов и антонимов, впервые показанная примере 3 раздела 14.5.

```
/* Ассоциативный список слов и антонимов для объектов типа string
*/
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    map<string, string> m;
    int i;

    m.insert(pair<string, string>("да", "нет"));
    m.insert(pair<string, string>("хорошо", "плохо"));
    m.insert(pair<string, string>("влево"), "вправо"));
    m.insert(pair<string, string>("вверх"), "вниз"));

    string s;
```

```
    cout << "Введите слово: ";
    cin >> s;

    map<string, string>::iterator p;

    p = m.find(s);
    if(p != m.end())
        cout << "Антоним: " << p->second;
    else
        cout << "Такого слова в ассоциативном списке нет\n";

    return 0;
}
```