第24章　标准模板库

近些年来，C++引入了许多新功能，其中被许多人认为最重要的一个功能是标准模板库（standard template library，STL）。STL的内容是在C++标准化过程中取得的主要成就之一，它提供了通用的模板化的类和函数，这些类和函数实现了许多为人们普遍使用的算法和数据结构。这些算法和数据结构支持矢量、列表、队列和堆栈。此外，STL还定义了各种访问这些算法和数据结构的例程。因为STL构造于模板类，所以这些算法和数据结构几乎可以被应用于任何数据类型。

STL是软件工程所使用的一些最复杂C++功能的综合。为了理解和使用STL，必须完全弄懂C++语言，包括指针、引用和模板。坦率地说，描述STL的模板语法看上去有点让人畏惧，尽管它实际上并没有看上去那么复杂。虽然本章的内容并不比其他章节的内容更难，但是如果你发现STL一开始有些令人困惑的话，也不要感到吃惊和沮丧。读者要耐心地学习本章的例子，不要让不熟悉的语法掩盖了STL的基础性和简明性。

本章的目的是对STL给出一个综述性的介绍，其中包括它的设计理念、组织、构成和使用时需要的编程技巧。因为STL是一个大型库，所以这里不可能介绍它的所有功能。然而，本书在第四部分提供了一个STL的完整参考。

本章还讲解了一种重要的类，即string。string类定义了一个字符串数据类型，该类使你可以像操作其他数据类型一样操作字符串：使用运算符。string类与STL的关系非常紧密。

24.1　STL概述

虽然标准模板库非常大而且它的语法也非常严格，但是一旦了解了它的构造方式和它所使用的元素，这个库使用起来非常简单。因此，在介绍代码范例之前，我们对STL进行一个概述性介绍。

标准模板库的核心有三种基础成分：容器、算法和迭代器。三者相互之间协同工作，从而为各种编程问题提供行之有效的解决方案。

24.1.1　容器

容器是存放其他对象的对象，容器有几种不同的类型。例如，vector类定义一个动态数组，deque创建一个双端队列，list提供一个线性列表。这些容器被称为顺序容器（sequence container），因为用STL的术语讲，一个序列是一个线性列表。除基本容器外，STL还定义了一些关联容器（associative container），这些容器使人们可以根据关键字进行有效的检索。例如，map容器利用惟一的关键字提供对值的访问。因此，一个map容器存储一个关键字/值对并允许根据给出的关键字检索一个值。

每个容器类都定义一组函数，这些函数可以被应用于该容器。例如，一个列表容器包括插入函数、删除函数和合并元素函数；一个堆栈容器包括用于推入和弹出值的函数。

24.1.2　算法

算法作用于容器，它们提供操作容器内容的方法。它们做的事情包括初始化、排序、搜索和转换容器内容。许多算法都可以在一个容器内对一定范围的元素进行操作。

24.1.3　迭代器

迭代器是一些对象，其操作与指针多少有些想像。迭代器使你可以对容器的内容进行循环，其循环方式类似于利用指针在数组中的循环。C++ 中提供了 5 种迭代器：

迭代器	允许的访问
随机访问迭代器	存储和检索值。元素可以被随机访问
双向迭代器	存储和检索值。可以向前和向后移动
前向迭代器	存储和检索值。只能向前移动
输入迭代器	检索但不存储值。只能向前移动
输出迭代器	存储但不检索值。只能向前移动

一般来说，可用访问能力较强的迭代器代替访问能力较弱的迭代器。例如，可以用前向迭代器代替输入迭代器。

迭代器的处理很像指针，可以对其进行递增和递减操作，也可以对其应用运算符 *。迭代器使用由各种容器定义的 iterator 类型声明。

STL 也支持反向迭代器。反向迭代器要么是双向的随机访问的迭代器，这种迭代器可以以相反的方向遍历一个序列。因此，如果一个反向迭代器指向一个序列的末尾，对该迭代器的递增操作将使其指向序列末尾的前一个元素。

当涉及模板说明中不同的迭代器类型时，本书将采用以下术语：

术语	含义
BiIter	双向迭代器
ForIter	前向迭代器
InIter	输入迭代器
OutIter	输出迭代器
RandIter	随机访问迭代器

24.1.4　其他 STL 元素

除容器、算法和迭代器之外，STL 还依赖其他几种标准组件，其中最主要的组件是分配器（allocator）、谓词（predicate）、比较函数和函数对象。

每个容器都定义一个分配器。分配器可管理一个容器的内存分配。默认的分配器是一个 allocator 类的对象，但是如果某些特殊应用需要的话，你也可以定义自己的分配器。大多数情况下，使用默认分配器就足够了。

个别算法和容器使用一个称为谓词的特殊类型的函数。谓词有两种变体：一元谓词和二元谓词。一元谓词有一个参数，二元谓词有两个参数。这些函数可以返回 true/false，而决定返回 true 或 false 的谓词条件则由你定义。在本章其余部分，当需要使用一元谓词函数时，则用 UnPred 类型标识；当需要使用二元谓词函数时，则用 BinPred 类型标识。在二元谓词中，参数总是按照第一、第二的顺序排列。无论是一元谓词还是二元谓词，其参数都将包含被该容器存储的对

象类型的值。

有些算法和类使用一种特殊的二元谓词，这种谓词可以比较两个元素。如果第一个参数小于第二个参数，那么比较函数返回true。比较函数将用类型Comp标识。

除各种STL类要求包含头文件外，C++标准库也包括<utility>和<functional>头文件，从而可以支持STL。例如，可以存放一对值的模板类pair就是在<utility>中定义的。本章后面还将用到pair。

<functional>中的模板有助于你构建定义operator()的对象。这些对象称为函数对象，而且在许多地方可以代替函数指针。<functional>内预先定义了几种函数对象，这些对象如下所示：

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

使用的最普遍的函数对象或许是less，它可以决定什么时候一个对象小于另一个对象。在后面介绍的STL算法中，我们可以用函数对象代替真正的函数指针。利用函数对象（而不是函数指针），可以使STL生成更有效的代码。

还有两个移植到STL的实体是绑定器和取反器（binders and negators）。一个绑定器可以把一个参数和一个函数对象绑定在一起，一个取反器将返回一个谓词的补码。最后一个术语是适配器（adaptor）。在STL术语中，适配器是用于进行事务转换的。例如，容器queue（该容器可以创建一个标准队列）是容器deque的一个适配器。

24.2 容器类

就像前面所讲的那样，容器实际上是存储数据的STL对象。STL定义的容器如表24.1所示。表24.1还列出了使用每一个容器时所必需的头文件。管理字符串的string类也是一个容器，我们将在本章后面进行讨论。

表24.1 STL定义的容器

容器	说明	需要的头文件
bitset	一组位	<bitset>
deque	一个双端队列	<deque>
list	一个线性列表	<list>
map	存储关键字/值对，其中每一个关键字只和一个值相关联	<map>
multimap	存储关键字/值对，其中每一个关键字可以和两个或多个值相关联	<map>
multiset	一个集合，其中每一个元素不必是惟一的	<set>
priority_queue	一个优先队列	<queue>
queue	一个队列	<queue>
set	一个集合，其中每一个元素必须是惟一的	<set>
stack	一个堆栈	<stack>
vector	一个动态数组	<vector>

由于模板类声明中一般的占位符类型名是任意的，所以容器类声明了这些类型的类型定义（typedef）版本，这样一来就使得类型名被具体化了。下面是一些最常用的typedef名：

size_type	一些整型
reference	对一个元素的引用
const_reference	对一个元素的 const 引用
iterator	一个迭代器
const_iterator	一个 const 迭代器
reverse_iterator	一个反向迭代器
const_reverse_iterator	一个 const 反向迭代器
value_type	存储在容器中的某个值的类型
allocator_type	分配器的类型
key_type	关键字的类型
key_compare	比较两个关键字的函数的类型
value_compare	比较两个值的函数的类型

24.3　一般的操作原理

虽然STL内部的操作非常复杂，但STL的使用却很简单。首先，你必须确定想要使用的容器类型，每种容器都有优点和缺点。例如，当需要一个随机访问的、类似于数组的对象而且不需要进行太多的插入和删除操作时，采用vector是非常适宜的。list虽然能够提供廉价的插入和删除，但是却以降低速度作为交换。map 提供一个关联容器，但却会导致附加的开销。

一旦选择了一个容器，就可以利用其成员函数往该容器中添加元素、访问或修改并删除这些元素。除 bitset 以外，一个容器在被添加元素时将自动扩大，在被删除元素时将自动缩小。

元素可以以许多不同的方式添加到容器中，或者是从容器中删除。例如，顺序容器（vector，list 和 deque）和关联容器（map，multimap，set 和 multiset）都提供一个称为 insert( )的成员函数，该函数可以将元素插入到一个容器中；它们还提供一个称为 erase( )的函数，该函数可以从一个容器中删除元素。此外，顺序容器还提供 push_back( )和 pop_back( )函数，它们分别把一个元素添加到容器末端或从容器末端删除。对于顺序容器而言，这些函数或许是最常用的添加或删除元素的方法。list 和 deque 容器还包括 push_front( )和 pop_front( )，它们能够从容器的起始处添加和删除元素。

在一个容器内部访问元素的最常用方法之一是通过迭代器。顺序容器和关联容器都提供了成员函数 begin( )和 end( )，这两个函数分别返回指向该容器起始位置和终止位置的迭代器。这些迭代器在访问容器内容时是非常实用的。例如，为了在一个容器中进行循环操作，可以利用 begin( )获得一个指向容器起始位置的迭代器，然后对其进行递增，直到它的值变为 end( )。

关联容器提供函数find( )，该函数被用于根据其关键字在一个关联容器中定位一个元素。因为关联容器把一个关键字和它的值相连接，所以 find( )是关联容器中大多数元素的定位方式。

因为 vector 是一个动态数组，所以它也支持用于访问其元素的标准数组索引语法。

一旦有了一个存放信息的容器，就可以利用一种或多种算法来使用它。这些算法不仅可以使你以某种规定的方式改变容器内容，还可以使你把一种顺序类型转换为另一种顺序类型。

在下面几节中，我们将介绍如何把这些方法应用于三个具有代表性的容器，即容器vector，list 和 map。一旦理解了这些容器的工作方式，那么在使用其他容器时也就不会有麻烦了。

24.4 vector容器

用途最多的容器或许是vector。vector类支持动态数组，这个数组可以根据需要扩大。就像你所知道的那样，在C++中，数组的大小在编译时是固定的。虽然到目前为止，这是一种最有效的数组实现方式，但是因为数组在运行时不能被调整以适应程序条件的改变，所以它也最受限制。而一个vector容器通过对内存进行按需分配可以解决这个问题。虽然vector容器是动态的，但仍然可以使用标准的数组下标符号访问它的元素。

vector的模板说明如下所示：

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

其中，T是被存储的数据类型，Allocator指定分配器，其默认值为标准分配器。vector具有下列构造函数：

```
explicit vector(const Allocator &a = Allocator( ) );
explicit vector(size_type num, const T &val = T ( ),
                const Allocator &a = Allocator( ));
vector(const vector<T, Allocator> &ob);
template <class InIter> vector(InIter start, InIter end,
                const Allocator &a = Allocator( ));
```

第一种形式构造一个空矢量。第二种形式构造一个具有num个元素、值为val的矢量。val的值可以是默认值。第三种形式构造一个矢量，该矢量含有的元素与ob包含的元素相同。第四种形式构造一个矢量，该矢量包含的元素限定在迭代器start和end指定的范围内。

为了获得最大的灵活性和可移植性，所有将被存储在一个vector容器中的对象都应该定义一个默认构造函数，它也应该定义<和==运算符。一些编译器可能要求定义其他的比较运算符（因为具体实现的不同，所以应该查阅相应的编译器文档以获取准确信息）。所有的内置类型都能自动满足这些要求。

虽然模板语法看上去比较复杂，但要声明一个vector容器并不难。下面列举了一些范例：

```
vector<int> iv;              // create zero-length int vector
vector<char> cv(5);          // create 5-element char vector
vector<char> cv(5, 'x');     // initialize a 5-element char vector
vector<int> iv2(iv);         // create int vector from an int vector
```

下面是为vector定义的比较运算符：

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

下标运算符[ ]也是为vector定义的。该运算符使你可以利用标准数组下标符号访问一个矢量的元素。

表24.2显示了vector定义的几种成员函数（记住，本书第四部分包含STL类的完整参考）。一些最常用的成员函数是size( )，begin( )，end( )，push_back( )，insert( )和erase( )。函数size( )可以返回当前矢量的大小。因为该函数使你可以在运行时确定一个矢量的大小，所以非常实用。记住，因为矢量可以根据需要增加其大小，所以一个矢量的大小必须在运行时确定，而不是在编译时确定。

函数begin( )返回一个指向矢量容器始端的迭代器，函数end( )返回一个指向矢量容器末端

的迭代器。就像前面讲述的那样，迭代器类似于指针，通过使用函数 begin( )和 end( )，可以获得一个指向矢量容器始端和末端的迭代器。

函数 push_back( )可以把一个值放置在矢量容器的末端。如果有必要，可以增加矢量容器的长度以容纳新元素。你也可以利用 insert( )在容器中间添加元素。此外，一个矢量容器还可以被初始化。无论如何，一旦矢量容器包含元素，就可以使用数组下标访问或修改这些元素，你还可以利用 erase( )从矢量容器中删除元素。

表 24.2　矢量容器定义的一些常用的成员函数

成员	说明
reference back( ); const_reference back( ) const;	返回对矢量容器中最后一个元素的引用
iterator begin( ); const_iterator begin( ) const;	返回指向矢量容器中第一个元素的迭代器
void clear( );	删除矢量容器中的所有元素
bool empty( ) const;	如果调用矢量容器为空，返回 true；否则，返回 false
iterator end( ); const_iterator end( ) const;	返回指向矢量容器中最后一个元素的迭代器
iterator erase(iterator *i*);	删除 i 所指向的元素并返回指向被删除元素后面的元素的迭代器
iterator erase(iterator start, iterator *end*);	删除 start ~ end 范围内的所有元素，返回指向最后一个被删除元素后面的元素的迭代器
reference front( ); const_reference front( ) const;	返回对矢量容器中第一个元素的引用
iterator insert(iterator *i*, const T *&val*);	在由 i 指定的元素前插入 val 并返回指向该元素的迭代器
void insert(iterator *i*, size_type *num*, const T & *val*)	在由 i 指定的元素前插入 num 个 val 的拷贝
template <class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	在由 i 指定的元素前插入由 start 和 end 定义的序列
reference operator[ ](size_type *i*) const; const_reference operator[ ](size_type *i*) const;	返回对由 i 指定的元素的引用
void pop_back( );	删除矢量容器中的最后一个元素
void push_back(const T *&val*);	把一个元素添加到矢量容器的末端，该元素的值由 val 指定
size_type size( ) const;	返回矢量容器中当前元素个数

下面是一个简短的例子，说明了一个矢量容器的基本操作。

```
// Demonstrate a vector.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;

int main()
{
  vector<char> v(10); // create a vector of length 10
  unsigned int i;

  // display original size of v
```

```
  cout << "Size = " << v.size() << endl;

  // assign the elements of the vector some values
  for(i=0; i<10; i++) v[i] = i + 'a';

  // display contents of vector
  cout << "Current Contents:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  cout << "Expanding vector\n";
  /* put more values onto the end of the vector,
     it will grow as needed */
  for(i=0; i<10; i++) v.push_back(i + 10 + 'a');

  // display current size of v
  cout << "Size now = " << v.size() << endl;

  // display contents of vector
  cout << "Current contents:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  // change contents of vector
  for(i=0; i<v.size(); i++) v[i] = toupper(v[i]);
  cout << "Modified Contents:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;
  return 0;
}
```

程序的输出如下所示：

```
Size = 10
Current Contents:
a b c d e f g h i j

Expanding vector
Size now = 20
Current contents:
a b c d e f g h i j k l m n o p q r s t

Modified Contents:
A B C D E F G H I J K L M N O P Q R S T
```

让我们仔细看一看这个程序。在main( )中创建了一个称为v的字符矢量，它最初的大小是10，也就是说，v最初包含10个元素。这一点可以通过调用成员函数size( )而得到证实。接下来，这10个元素被初始化为字符a～j，然后显示v的内容。我们注意到，这里使用了标准数组下标符号。接下来，利用push_back( )函数往v的末端又添加了10元素，为了容纳这些新元素，该操作扩大了v的容量。就像输出所示的，添加新元素后，矢量的大小增大到20。最后，利用标准下标符号改变了v中的元素值。

这个程序中还有一点有趣之处：在显示v中内容的循环内使用了它们的目标值v.size( )。可见，同数组相比，矢量所具备的优点之一是有可能得知矢量的当前大小。可以想像，在各种情况下，这都是很实用的。

24.4.1 通过迭代器访问矢量

就像读者已经知道的那样，在C++中，数组和指针紧密地联系在一起。一个数组既可以通过下标访问，也可以通过指针来访问。在STL中与这种联系相应的是矢量和迭代器之间的联系。矢量的成员既可以通过下标访问，也可以通过迭代器访问。下面的例子说明了访问方式。

```
// Access the elements of a vector through an iterator.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;

int main()
{
  vector<char> v(10); // create a vector of length 10
  vector<char>::iterator p; // create an iterator
  int i;

  // assign elements in vector a value
  p = v.begin();
    i = 0;
  while(p != v.end()) {
    *p = i + 'a';
    p++;
    i++;
  }

  // display contents of vector
  cout << "Original contents:\n";
  p = v.begin();
  while(p != v.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  // change contents of vector
  p = v.begin();
  while(p != v.end()) {
    *p = toupper(*p);
    p++;
  }

  // display contents of vector
  cout << "Modified Contents:\n";
  p = v.begin();
  while(p != v.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;
  return 0;
}
```

上面程序的输出如下所示：

```
Original contents:
a b c d e f g h i j
Modified Contents:
A B C D E F G H I J
```

我们注意到在这个程序中迭代器p的声明方式。类型iterator由容器类定义，因此，为了获得一个特殊容器的迭代器，可以使用类似于该例子中所示的声明：简单地用容器名限定iterator。在这个程序中，利用程序函数begin( )将p初始化为指向矢量的起始位置。该函数返回一个指向矢量起始端的迭代器，因为这个迭代器可以在需要时递增，所以该迭代器能够在运行时访问矢量元素。这个过程直接与访问数组元素的指针相对应。为了确定何时到达矢量末端，可以使用成员函数end( )。该函数返回一个迭代器，该迭代器所指的位置位于矢量中最后一个元素之后。因此，当p等于v.end( )时，说明到达了矢量末端。

24.4.2 在矢量中插入和输出元素

除了可以把新值放置在矢量末端之外，还可以用insert( )函数把元素插入到矢量的中间，另外还可以用erase( )函数删除元素。下面的程序演示了insert( )和erase( )的使用方法。

```
// Demonstrate insert and erase.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<char> v(10);
  vector<char> v2;
  char str[] = "<Vector>";
  unsigned int i;
  // initialize v
  for(i=0; i<10; i++) v[i] = i + 'a';

  // copy characters in str into v2
  for(i=0; str[i]; i++) v2.push_back(str[i]);

  // display original contents of vector
  cout << "Original contents of v:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  vector<char>::iterator p = v.begin();
  p += 2; // point to 3rd element

  // insert 10 X's into v
  v.insert(p, 10, 'X');

  // display contents after insertion
  cout << "Size after inserting X's = " << v.size() << endl;
  cout << "Contents after insert:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";
```

```
  // remove those elements
  p = v.begin();
  p += 2; // point to 3rd element
  v.erase(p, p+10); // remove next 10 elements

  // display contents after deletion
  cout << "Size after erase = " << v.size() << endl;
  cout << "Contents after erase:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  // Insert v2 into v
  v.insert(p, v2.begin(), v2.end());
  cout << "Size after v2's insertion = ";
  cout << v.size() << endl;
  cout << "Contents after insert:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  return 0;
}
```

上面程序的输出如下所示：

```
Original contents of v:
a b c d e f g h i j

Size after inserting X's = 20
Contents after insert:
a b X X X X X X X X X c d e f g h i j

Size after erase = 10
Contents after erase:
a b c d e f g h i j

Size after v2's insertion = 18
Contents after insert:
a b < V e c t o r > c d e f g h i j
```

这个程序使用了两种形式的 insert( )。第一次使用时是将 10 个 X 插入到 v 中，第二次是将第二个矢量（v2）的内容插入到 v 中。最有趣的是第二次使用 insert( )，这一次函数带有三个迭代器参数：第一个参数指定一个指针，该指针指向调用容器中的插入位置；后面两个指针分别指向被插入元素序列的始端和末端。

24.4.3　在矢量中存储类对象

虽然前面的例子在矢量中存储的对象都是内置类型，但是矢量却不仅只限于此。矢量可以存储任何类型的对象，其中包括你所创建的那些类。下面的例子使用了一个存储对象的矢量，该矢量用于存放一周之内的每日最高气温。可以看出，DailyTemp 定义了一个默认构造函数，而且程序还提供了重载的<和==。记住，由于编译器实现 STL 的方式各不相同，因此根据你所使用的编译器，或许需要定义这些（或其他的）比较运算符。

```
// Store a class object in a vector.
#include <iostream>
```

```
#include <vector>
#include <cstdlib>
using namespace std;
class DailyTemp {
  int temp;
public:
  DailyTemp() { temp = 0; }
  DailyTemp(int x) { temp = x; }

  DailyTemp &operator=(int x) {
    temp = x; return *this;
  }

  double get_temp() { return temp; }
};

bool operator<(DailyTemp a, DailyTemp b)
{
  return a.get_temp() < b.get_temp();
}

bool operator==(DailyTemp a, DailyTemp b)
{
  return a.get_temp() == b.get_temp();
}

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

  for(i=0; i<7; i++)
    v.push_back(DailyTemp(60 + rand()%30));

  cout << "Fahrenheit temperatures:\n";
  for(i=0; i<v.size(); i++)
    cout << v[i].get_temp() << " ";

  cout << endl;

  // convert from Fahrenheit to Centigrade
  for(i=0; i<v.size(); i++)
    v[i] = (int)(v[i].get_temp()-32) * 5/9 ;

  cout << "Centigrade temperatures:\n";
  for(i=0; i<v.size(); i++)
    cout << v[i].get_temp() << " ";

  return 0;
}
```

下面是该程序的输出范例：

```
Fahrenheit temperatures:
71 77 64 70 89 64 78
Centigrade temperatures:
21 25 17 21 31 17 25
```

矢量提供了强大的功能性、安全性和灵活性，但是却不如一般的数组那样高效。因此，对大多数编程工作而言，数组仍将是你的首选，然而你也应该时刻关注使用矢量带来的益处。

24.5 list 容器

list类支持双向线性列表。和支持随机访问的矢量不同，列表只能被顺序访问。由于列表是双向的，所以它们既可以按照从前到后的顺序访问，也可以按照从后到前的顺序访问。

list 具有下面的模板说明：

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

其中，T是列表中存储的数据类型。分配器由 Allocator指定，其默认值为标准分配器。它具有下列构造函数：

```
explicit list(const Allocator &a = Allocator( ) );
explicit list(size_type num, const T &val = T ( ),
              const Allocator &a = Allocator( ));
list(const list<T, Allocator> &ob);
template <class InIter>list(InIter start, InIter end,
              const Allocator &a = Allocator( ));
```

第一种形式构造一个空列表；第二种形式构造一个列表，该列表含有 num个元素，元素的值为 val，该值可以是默认值；第三种形式构造一个列表，该列表包含的元素与 ob 的相同；第四种形式构造一个列表，该列表包含的元素在迭代器 start 和 end 指定的范围之内。

下面是为列表定义的比较运算符：

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

表24.3列出了一些常用的list成员函数。像矢量一样，可以利用push_back( )函数把元素放入一个列表。还可以利用push_front( )把元素放入列表的前面。也可以利用 insert( )把元素插入到列表的中间。利用splice( )可以把两个列表连接起来，而利用merge( )可以把一个列表合并入另一个列表。

表 24.3　一些常用的 list 成员函数

成员	说明
reference back( ); const_reference back( ) const;	返回对列表中最后一个元素的引用
iterator begin( ); const_iterator begin( ) const;	返回指向列表中第一个元素的迭代器
void clear( );	删除列表中的所有元素
bool empty( ) const;	如果调用列表为空，返回 true；否则，返回 false
iterator end( ); const_iterator end( ) const;	返回一个指向列表末尾的迭代器
iterator erase(iterator *i*);	删除 i 所指的元素并返回指向被删除元素后面那个元素的迭代器
iterator erase(iterator start, iterator *end*);	删除start ~ end范围内的元素并返回指向被删除的最后一个元素后面的那个元素的迭代器
reference front( ); const_reference front( ) const;	返回对列表中第一个元素的引用

（续表）

成员	说明
iterator insert(iterator *i*, const T &*val*);	在i指定的元素之前插入val并返回指向该元素的迭代器
void insert(iterator *i*, size_type *num*, const T &*val*)	在i指定的元素之前插入num个val的副本
template <class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	在i指定的元素之前插入由start和end定义的序列
void merge(list<T, Allocator> &*ob*); template <class Comp> void merge(list<T, Allocator> &*ob*, Comp *cmpfn*);	将ob中的有序列表并入有序的调用列表中，结果仍是一个有序列表。合并之后，ob中包含的列表为空。在第二种形式中，可以指定比较函数，从而确定何时一个元素小于另一个元素
void pop_back( );	删除列表中的最后一个元素
void pop_front( );	删除列表中的第一个元素
void push_back(const T &*val*);	把一个元素添加到列表末尾，该元素的值由val指定
void push_front(const T &*val*);	把一个元素添加到列表开始处，该元素的值由val指定
void remove(const T &*val*);	从列表中删除值为val的元素
void reverse( );	反转调用列表
size_type size( ) const;	返回列表中的当前元素个数.

void sort( );
template <class Comp>
void sort(Comp *cmpfn*);
	给列表排序。第二种形式利用比较函数cmpfn给列表排序以确定什么时候一个元素小于另一个元素

void splice(iterator *i*, list<T, Allocator> &*ob*);	将ob的内容按照i所指的位置插入到调用列表中。操作完成后，ob将为空
void splice(iterator *i*, list<T, Allocator> &*ob*, iterator *el*);	从ob中删除el指向的元素并将该元素按i所指的位置存储到调用列表中
void splice(iterator *i*, list<T, Allocator> &*ob*, iterator *start*, iterator *end*);	从ob中删除start～end范围内的元素并将这些元素按照i所指的位置存放到调用列表中

为了获得最大程度的灵活性和可移植性，所有存储在列表中的对象都应定义一个默认构造函数，此外，还应该定义一个<运算符和其他比较运算符。存储在列表中的针对某个对象的精确要求随编译器的不同而不同，所以需要查阅相应的编译器文档。

下面是一个使用列表的简单例子。

```
// List basics.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst; // create an empty list
  int i;
```

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

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

  cout << "Contents: ";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  // change contents of list
  p = lst.begin();
  while(p != lst.end()) {
    *p = *p + 100;
    p++;
  }

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

  return 0;
}
```

程序的输出如下所示：

```
Size = 10
Contents: 0 1 2 3 4 5 6 7 8 9

Contents modified: 100 101 102 103 104 105 106 107 108 109
```

该程序创建了一个整型列表。程序首先创建了一个空的 list 对象，接下来往该列表中放入了 10 个整型数。这是用 push_back( )函数完成的，push_back( )函数把每一个新值放入已有的列表末尾。然后，程序显示该列表的大小和列表的内容。列表通过一个迭代器显示，其代码如下：

```
list<int>::iterator p = lst.begin();
while(p != lst.end()) {
  cout << *p << " ";
  p++;
}
```

其中，迭代器 p 被初始化为指向列表的起始位置。每经过一次循环，p 的值就增加 1，从而指向下一个元素。当 p 指向列表末尾时，循环结束。上面的代码从本质上说与利用迭代器对一个矢量进行循环操作是相同的。这样的循环在 STL 代码中很常见，可以利用同样的结构访问不同类型的容器的行为是 STL 所具能力的一部分。

24.5.1　了解 end( )函数

现在来看一看容器函数 end( )具有的多少有些出人意料的属性。end( )不返回指向容器中最

后一个元素的指针，相反，它返回一个指向最后一个元素后面位置的指针。因此，end( ) -1才指向容器中最后一个元素。利用这个特征可以编写一些非常高效的算法，从而可以利用迭代器对容器中的所有元素进行循环，其中包括最后一个元素。当迭代器的值与end( )的返回值相同时，说明访问了所有元素。然而，你必须牢记这个特征，因为它看上去有点与直觉相反。例如，看一看下面的程序，该程序分别以正向和反向方式显示一个列表。

```
// Understanding end().
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst; // create an empty list
  int i;

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

  cout << "List printed forwards:\n";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  cout << "List printed backwards:\n";
  p = lst.end();
  while(p != lst.begin()) {
    p--; // decrement pointer before using
    cout << *p << " ";
  }

  return 0;
}
```

程序的输出如下所示：

```
List printed forwards:
0 1 2 3 4 5 6 7 8 9

List printed backwards:
9 8 7 6 5 4 3 2 1 0
```

以正向方式显示的列表的代码与我们一直在用的代码相同，我们特别要关注的是以反向方式显示的列表的代码。通过使用end( )函数，可以把迭代器p初始化为指向列表末尾。因为end( )返回一个指向列表最后一个对象后面的对象的迭代器，所以p在使用之前必须被减1，这就是在执行循环内的cout语句之前而不是之后对p进行递减操作的原因所在。记住，end( )返回的指针指的不是列表中的最后一个对象，它指的是最后一个对象后面的元素。

24.5.2 push_front( )与push_back( )

可以通过向列表开始处或列表末尾添加元素来构建一个列表。到目前为止，我们已经利

用push_back( )往列表末尾添加过元素。为了往列表开始处添加元素，可以使用push_front( )。例如：

```
/* Demonstrating the difference between
   push_back() and push_front(). */
#include <iostream>
#include <list>
using namespace std;

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

  for(i=0; i<10; i++) lst1.push_back(i);
  for(i=0; i<10; i++) lst2.push_front(i);

  list<int>::iterator p;

  cout << "Contents of lst1:\n";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  cout << "Contents of lst2:\n";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

这个程序的输出如下所示：

```
Contents of lst1:
0 1 2 3 4 5 6 7 8 9

Contents of lst2:
9 8 7 6 5 4 3 2 1 0
```

因为lst2是通过往列表开始处添加元素构建的，所以结果列表的顺序与通过往列表末尾放置元素而构建的lst1相反。

24.5.3 列表排序

通过调用sort( )成员函数可以给列表排序。下面的程序创建一个随机整数列表，然后对这个列表进行排序。

```
// Sort a list.
#include <iostream>
```

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

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

  // create a list of random integers
  for(i=0; i<10; i++)
    lst.push_back(rand());

  cout << "Original contents:\n";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }

  cout << endl << endl;

  // sort the list
  lst.sort();

  cout << "Sorted contents:\n";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

下面是该程序的输出范例：

```
Original contents:
41 18467 6334 26500 19169 15724 11478 29358 26962 24464

Sorted contents:
41 6334 11478 15724 18467 19169 24464 26500 26962 29358
```

24.5.4 合并两个列表

一个有序列表可以和另一个有序列表合并，结果是一个包含两个列表内容的有序列表。新的列表为调用列表，而第二个列表则变为空列表。下面的例子合并了两个列表。第一个列表包含0～9之间的偶数，第二个列表包含0～9之间的奇数，这两个列表合并后产生序列0 1 2 3 4 5 6 7 8 9。

```
// Merge two lists.
#include <iostream>
#include <list>
using namespace std;

int main()
```

```
{
  list<int> lst1, lst2;
  int i;
  for(i=0; i<10; i+=2) lst1.push_back(i);
  for(i-1; i<11; i+=2) lst2.push_back(i);

  cout << "Contents of lst1:\n";
  list<int>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  cout << "Contents of lst2:\n";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  // now, merge the two lists
  lst1.merge(lst2);
  if(lst2.empty())
    cout << "lst2 is now empty\n";

  cout << "Contents of lst1 after merge:\n";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

这个程序的输出如下所示：

```
Contents of lst1:
0 2 4 6 8

Contents of lst2:
1 3 5 7 9

lst2 is now empty
Contents of lst1 after merge:
0 1 2 3 4 5 6 7 8 9
```

关于这个例子，还有一件事情需要注意，这就是empty( )函数的使用。如果调用容器为空，该函数返回 true。因为 merge( )将从被合并的列表中删除所有元素，所以该列表在合并操作完成后将变为空列表，这一点可以从程序的输出中得到证实。

24.5.5 在列表中存储类对象

下面的例子利用列表存储myclass类型的对象。注意,<,>,!=和==为myclass类型的对象而重载(对某些编译器而言,不必定义所有这些运算符;而对其他编译器来说,则需要定义附加的运算符)。STL使用这些函数确定容器中的对象次序和是否相等。即使某列表不是有序列表,当对其进行查询、排序或合并时,仍然需要一种方法来比较列表的元素。

```
// Store class objects in a list.
#include <iostream>
#include <list>
#include <cstring>
using namespace std;

class myclass {
  int a, b;
  int sum;
public:
  myclass() { a = b = 0; }
  myclass(int i, int j) {
    a = i;
    b = j;
    sum = a + b;
  }
  int getsum() { return sum; }

  friend bool operator<(const myclass &o1,
                        const myclass &o2);
  friend bool operator>(const myclass &o1,
                        const myclass &o2);
  friend bool operator==(const myclass &o1,
                        const myclass &o2);
  friend bool operator!=(const myclass &o1,
                        const myclass &o2);
};

bool operator<(const myclass &o1, const myclass &o2)
{
  return o1.sum < o2.sum;
}

bool operator>(const myclass &o1, const myclass &o2)
{
  return o1.sum > o2.sum;
}

bool operator==(const myclass &o1, const myclass &o2)
{
  return o1.sum == o2.sum;
}

bool operator!=(const myclass &o1, const myclass &o2)
{
  return o1.sum != o2.sum;
}
```

```
int main()
{
int i;

  // create first list
  list<myclass> lst1;
  for(i=0; i<10; i++) lst1.push_back(myclass(i, i));

  cout << "First list: ";
  list<myclass>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << p->getsum() << " ";
    p++;
  }
  cout << endl;

  // create a second list
  list<myclass> lst2;
  for(i=0; i<10; i++) lst2.push_back(myclass(i*2, i*3));

  cout << "Second list: ";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << p->getsum() << " ";
    p++;
  }
  cout << endl;

  // now, merget lst1 and lst2
  lst1.merge(lst2);

  // display merged list
  cout << "Merged list: ";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << p->getsum() << " ";
    p++;
  }

  return 0;
}
```

这个程序创建两个 myclass 对象列表并显示每一个列表的内容。然后，程序对这两个列表进行合并并显示合并结果。该程序的输出如下所示：

```
First list: 0 2 4 6 8 10 12 14 16 18
Second list: 0 5 10 15 20 25 30 35 40 45
Merged list: 0 0 2 4 5 6 8 10 10 12 14 15 16 18 20 25 30 35 40 45
```

24.6　map 容器

map 类支持一个关联容器，在这种容器中，那些具有惟一性的关键字被映射为相应的值。从本质上讲，一个关键字只是给某个值取的名字。一旦存储了一个值，就可以利用它的关键字对其进行检索。因此，从最普通的意义上说，一个映射是一个关键字/值对的列表。映射的功

能是使你可以根据给定的关键字查找一个值。例如，你可以定义一个映射，该映射将人名用做关键字，将电话号码用做值。现在，关联容器在编程中正在变得越来越流行。

我们在前面曾经提到，一个映射只能存放具有惟一性的关键字而不能存放重复的关键字。为了创建能够存放非惟一性关键字的映射，可以使用multimap。

map容器具有以下模板说明：

```
template <class Key, class T,.class Comp = less<Key>,
        class Allocator = allocator<pair<const key, T> > class map
```

其中，Key是关键字的数据类型，T是被存储（被映射）的值的数据类型，Comp是对两个关键字进行比较的函数，它默认为标准的less( )实用函数对象，Allocator是分配器（默认为allocator）。

一个map容器具有以下构造函数：

```
explicit map(const Comp &cmpfn = Comp( ),
        const Allocator &a = Allocator( ) );
map(const map<Key, T, Comp, Allocator> &ob);
template <class InIter> map(InIter start, InIter end,
        const Comp &cmpfn = Comp( ), const Allocator &a = Allocator( ));
```

第一种形式构造一个空映射；第二种形式构造一个与ob含有同样元素的映射；第三种形式构造一个映射，它包含的元素在迭代器start和end指定的范围之内，由cmpfn指定的函数（如果有的话）确定该映射的顺序。

一般来说，任何用做关键字的对象都应该定义一个默认构造函数并应重载<运算符和所有其他必需的比较运算符。对于不同的编译器，还会有一些专门的要求。

以下是针对映射定义的比较运算符。

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

表24.4列出了几种map成员函数，其中，key_type是关键字的类型，value_type表示pair <Key, T>。

表24.4 几个常用的map成员函数

成员	说明
iterator begin( ); const_iterator begin( ) const;	返回一个指向映射中第一个元素的迭代器
void clear( );	删除映射中的所有元素
size_type count(const key_type &k) const;	返回映射中出现k的次数（1或0）
bool empty( ) const;	如果映射为空，返回true；否则，返回false
iterator end( ); const_iterator end( ) const;	返回一个指向列表末端的迭代器
void erase(iterator *i*); void erase(iterator *start*, iterator *end*);	删除i所指的元素 删除start ~ end范围内的元素
size_type erase(const key_type &*k*)	从映射中删除关键字值为k的元素
iterator find(const key_type &*k*); const_iterator find(const key_type &*k*) const;	返回一个指向特定关键字的迭代器。如果没有发现该关键字，则返回一个指向映射末端的迭代器

（续表）

成员	说明
iterator insert(iterator *i*, const value_type &*val*);	在 i 所指的位置或在 i 所指的元素后面插入 val，然后返回一个指向该元素的指针
template <class InIter> void insert(InIter *start*, InIter *end*)	插入一定范围内的元素
pair<iterator, bool> insert(const value_type &*val*);	把 val 插入到调用映射中。返回一个指向该元素的迭代器。只有在还不存在该元素的情况下才插入它。如果插入了这个元素，则返回pair<iterator, true>；否则，返回pair<iterator,false>
mapped_type & operator[ ](const key_type &*i*)	返回一个对i指定的元素的引用。如果该元素不存在，就插入它
size_type size( ) const;	返回列表中的当前元素个数

关键字 / 值对作为 pair 类型的对象存储在一个映射中，其模板说明如下：

```
template <class Ktype, class Vtype> struct pair {
  typedef Ktype first_type; // type of key
  typedef Vtype second_type; // type of value
  Ktype first; // contains the key
  Vtype second; // contains the value

  // constructors
  pair();
  pair(const Ktype &k, const Vtype &v);
  template<class A, class B> pair(const<A, B> &ob);
}
```

就像注释语句说明的那样，first 中的值包含关键字，second 中的值包含与该关键字相关联的值。

可以利用任何一个 pair 的构造函数，或利用 make_pair( )来构造一个关键字 / 值对。后者根据用做参数的数据类型构建一个 pair 对象。make_pair( )是一个通用函数，其原型如下：

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

可以看到，该函数返回一个由 *Ktype* 和 *Vtype* 指定的类型的值组成的 pair 对象。make_pair( ) 的优点是被存储的对象类型由编译器自动确定，而不是由编程人员显式地指定。

下面的程序说明了使用映射的基本方法。该程序存储关键字 / 值对，这些关键字 / 值对显示了大写字母和其 ASCII 码之间的映射关系。因此，这里的关键字是一个字符，值是一个整数。该程序存储的关键字 / 值对是：

```
A       65
B       66
C       67
```

等。一旦存储了这些关键字 / 值对，就会被提示输入一个关键字（即：A 和 Z 之间的一个字母）并显示该字母的 ASCII 码。

```
// A simple map demonstration.
#include <iostream>
#include <map>
```

```
using namespace std;

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

  // put pairs into map
  for(i=0; i<26; i++) {
    m.insert(pair<char, int>('A'+i, 65+i));
  }

  char ch;
  cout << "Enter key: ";
  cin >> ch;

  map<char, int>::iterator p;

  // find value given key
  p = m.find(ch);
  if(p != m.end())
    cout << "Its ASCII value is " << p->second;
  else
    cout << "Key not in map.\n";

  return 0;
}
```

我们注意到构建关键字/值对的pair模板类的使用方法。pair指定的数据类型必须与那些被插入关键字/值对的映射相匹配。

一旦用关键字和值对映射进行了初始化，就可以利用find( )函数根据给定的关键字对其进行检索。find( )将返回一个指向匹配元素的迭代器，如果没有找到相应的关键字，该函数将返回一个指向映射末端的迭代器。当找到一个匹配元素时，与关键字相关的值被包含在pair的second成员中。

在前面的例子中，关键字/值对是利用pair<char, int>显式地构建的。虽然这种方法没有任何错误，但是使用make_pair( )将更加简单，该函数根据用做参数的数据类型构建一个pair对象。例如，还以前面的程序为例，下面这行代码也将把关键字/值对插入到m中。

```
m.insert(make_pair((char)('A'+i), 65+i));
```

其中，当i被添加到“A”时，强制转换到char的操作需要重载自动转换为int的操作。否则，类型确定将是自动的。

24.6.1 在映射中存储类对象

可以利用映射存储你创建的类型的对象。例如，下面的程序创建一个简单的电话号码簿，也就是说，创建一个名字和电话号码的映射。为此，该程序创建了两个类，分别称为name和number。因为一个映射维护着一个关键字的排序列表，所以该程序也为name类型的对象定义了运算符<。一般来说，你必须为所有用做关键字的类定义<运算符（有些编译器可能需要定义附加的比较运算符）。

```
// Use a map to create a phone directory.
```

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

class name {
  char str[40];
public:
  name() { strcpy(str, ""); }
  name(char *s) { strcpy(str, s); }
  char *get() { return str; }

};

// Must define less than relative to name objects.
bool operator<(name a, name b)
{
  return strcmp(a.get(), b.get()) < 0;
}

class phoneNum {
  char str[80];
public:
  phoneNum() { strcmp(str, ""); }
  phoneNum(char *s) { strcpy(str, s); }
  char *get() { return str; }
};

int main()
{
  map<name, phoneNum> directory;

  // put names and numbers into map
   directory.insert(pair<name, phoneNum>(name("Tom"),
                    phoneNum("555-4533")));
   directory.insert(pair<name, phoneNum>(name("Chris"),
                    phoneNum("555-9678")));
   directory.insert(pair<name, phoneNum>(name("John"),
              phoneNum("555-8195")));
   directory.insert(pair<name, phoneNum>(name("Rachel"),
                    phoneNum("555-0809")));

  // given a name, find number
  char str[80];
  cout << "Enter name: ";
  cin >> str;

  map<name, phoneNum>::iterator p;

  p = directory.find(name(str));
  if(p != directory.end())
    cout << "Phone number: " << p->second.get();
  else
    cout << "Name not in directory.\n";

  return 0;
}
```

下面是该程序的一个运行范例：

```
Enter name: Rachel
Phone number: 555-0809.
```

在这个程序中，映射中的每一个条目都是一个字符数组，该数组存放一个以Null结束的字符串。我们在本章后面将利用标准的string类型编写这个程序，这将是一种更简单的方法。

24.7 算法

就像前面讲到的那样，算法针对容器起作用。尽管所有容器都对其自身的基本操作提供支持，但是标准算法还支持更广泛、更复杂的操作。此外，标准算法还可以使你同时操作两种不同类型的容器。为了访问STL算法，必须在程序中包括<algorithm>。

STL定义了很多算法，表24.5给出这些算法的一览。所有算法都是模板函数，这意味着可以把算法应用于任何类型的容器。本书将在第四部分介绍STL中的算法，下面几节将演示一些具有代表性的范例。

表24.5 STL算法

算法	用途
adjacent_find	搜索一个序列中邻近的匹配元素并返回一个指向第一个匹配元素的迭代器
binary_search	对一个有序序列进行二分查找
copy	复制一个序列
copy_backward	除了它首先从序列末端移动元素外，其功能与copy( )相同
count	返回序列中的元素个数
count_if	返回序列中满足某种谓词条件的元素个数
equal	确定两个范围是否相同
equal_range	返回一个序列范围，在该范围中可以插入一个元素而且不破坏序列顺序
fill和fill_n	用指定的值填充一个范围
find	在某个范围内搜索一个值并返回一个指向第一次出现该值的位置的迭代器
find_end	在某个范围内搜索一个子序列并返回一个指向子序列末端的迭代器
find_first_of	在子序列中查找第一个与某个范围内的一个元素相匹配的元素
find_if	在一定的范围内搜索一个元素，该元素使用户定义的一个一元谓词返回true
for_each	将一个函数应用子某个范围内的元素
generate和generate_n	把一个生成器函数返回的值赋给某个范围内的元素
includes	确定一个序列是否包括另一个序列中的所有元素
inplace_merge	把一个范围和另一个范围合并在一起。这两个序列必须按照递增的顺序存储，其结果序列也将被存储
iter_swap	交换由两个迭代器参数所指的值
lexicographical_compare	按照字母顺序对两个序列进行比较
lower_bound	查找序列中小于指定值的第一个元素位置
make_heap	从一个序列中构建一个堆
max	返回两个值中的最大值
max_element	返回一个指向某范围内最大元素的迭代器

（续表）

算法	用途
merge	合并两个有序序列并将合并结果放入第三个序列
min	返回两个值中的最小值
min_element	返回一个指向某范围内最小元素的迭代器
mismatch	在两个序列中查找第一个失配元素并返回指向这两个元素的迭代器
next_permutation	构建序列的下一种排列
nth_element	把所有小于E 的元素排在该元素之前，把所有大于E 的元素排在该元素之后
partial_sort	在一个范围内进行排序
partial_sort_copy	在一个范围内进行排序并复制适合放入结果序列中的所有元素
partition	排列一个子序列，将所有使谓词返回true的元素排在使该谓词返回false的元素之前
pop_heap	交换第一个（first）和倒数第二个（last - 1）元素，然后重新构建这个堆
prev_permutation	构建子序列的前一种排列
push_heap	把一个元素压入一个堆的末端
random_shuffle	把一个序列随机化
remove, remove_if, remove_copy 和 remove_copy_if	删除指定范围内的元素
replace, replace_copy, replace_if 和 replace_copy_if	替换一个范围内的元素
reverse 和 reverse_copy	颠倒一个范围的元素顺序
rotate 和 rotate_copy	循环左移一个范围内的元素
search	搜索一个序列中的子序列
search_n	搜索一个具有指定的相似元素个数的序列
set_difference	产生一个序列，该序列包含两个有序集合之间的差集
set_intersection	产生一个序列，该序列包含两个有序集合之间的交集
set_symmetric_difference	产生一个序列，该序列包含两个有序集合之间的对称差集
set_union	产生一个序列，该序列包含两个有序集合之间的并集
sort	对一个范围进行排序
sort_heap	对指定范围内的堆进行排序
stable_partition	对一个序列进行排列，将所有使谓词返回true的元素排在使谓词返回false的元素之前。这是一种稳定分割，也就是说该序列的相对顺序保持不变
stable_sort	对一个范围进行排序。这是一种稳定排序，也就是说，相等的元素不再被重新排列
swap	交换两个值
swap_ranges	交换一个范围内的元素
transform	将一个函数应用于一定范围内的元素并将产生的结果存入一个新序列
unique 和 unique_copy	删除一个范围内的重复元素
upper_bound	查找一个序列中大于某个值的最后一个元素

24.7.1 计数

对一个序列而言，最基本的操作之一是统计其元素个数。为此，可以使用count( )或者count_if( )，它们的一般形式如下：

```
template <class InIter, class T>
    ptrdiff_t count(InIter start, InIter end, const T &val);
template <class InIter, class UnPred>
    ptrdiff_t count_if(InIter start, InIter end, UnPred pfn);
```

类型ptrdiff_t被定义为某种整型形式。

count( )算法返回序列中与val相匹配的元素个数，该序列始于start，终于end。count_if( )算法返回序列中使一元谓词pfn返回true的元素个数，该序列始于start，终于end。

下面的程序演示了count( )。

```
// Demonstrate count().
#include <iostream>
#include <vector>
#include <cstdlib>
#include <algorithm>
using namespace std;

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

  for(i=0; i < 10; i++) {
   if(rand() % 2) v.push_back(true);
   else v.push_back(false);
  }

  cout << "Sequence:\n";
  for(i=0; i<v.size(); i++)
    cout << boolalpha << v[i] << " ";
  cout << endl;

  i = count(v.begin(), v.end(), true);
  cout << i << " elements are true.\n";

  return 0;
}
```

上面程序的输出如下：

```
Sequence:
true true false false true false false false false false
3 elements are true.
```

这个程序先是创建一个矢量，该矢量由随机生成的true和false值组成，接下来用count( )统计true值的个数。

下面的程序演示了count_if( )。该程序创建一个包含数字1～19的矢量，然后统计能够被3整除的元素个数。为了达到这个目的，该程序创建了一个称为dividesBy3( )的一元谓词，如

果谓词的参数能够被 3 整除，该谓词返回 true。

```
// Demonstrate count_if().
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

/* This is a unary predicate that determines
   if number is divisible by 3. */
bool dividesBy3(int i)
{
  if((i%3) == 0) return true;

  return false;
}

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

  for(i=1; i < 20; i++) v.push_back(i);

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

  i = count_if(v.begin(), v.end(), dividesBy3);
  cout << i << " numbers are divisible by 3.\n";

  return 0;
}
```

这个程序的输出如下所示：

```
Sequence:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
6 numbers are divisible by 3.
```

注意一元谓词 dividesBy3( )是怎样编码的。所有的一元谓词都接受一个参数，该参数的类型与存储在相应容器中的对象的类型相同，谓词据此参数进行操作。此外，谓词必须根据这个对象返回 true 或 false。

24.7.2　删除与替换元素

有时，生成一个只由最初序列中的某些元素组成的新序列是很有用的，有一个称为 remove_copy( )的算法可以完成这个任务。该算法的一般形式如下：

```
template <class InIter, class OutIter, class T>
  OutIter remove_copy(InIter start, InIter end,
                      OutIter result, const T &val);
```

remove_copy( )算法复制指定范围内的元素并删除值为 val 的那些元素，它把结果放入由 result 指向的序列并返回一个指向结果末端的迭代器。另外，输出容器必须足够大以便能够存放

结果。

当制作一个副本时，为了用另一个元素替代序列中的某个元素，可使用replace_copy( )。该算法的一般形式如下所示：

```
template <class InIter, class OutIter, class T>
  OutIter replace_copy(InIter start, InIter end,
                       OutIter result, const T &old, const T &new);
```

replace_copy( )算法从指定的范围复制元素并用new替代值为old的元素。该算法把结果放入由result指向的序列并返回一个指向结果末端的迭代器。另外，输出容器必须足够大，以便能够存放结果。

下面的程序演示了remove_copy( )和replace_copy( )的使用方法。该程序创建一个字符序列，然后删除该序列中的所有空格，之后再用冒号替换所有空格。

```
// Demonstrate remove_copy and replace_copy.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  char str[] = "The STL is power programming.";
  vector<char> v, v2(30);
  unsigned int i;

  for(i=0; str[i]; i++) v.push_back(str[i]);

  // **** demonstrate remove_copy ****
  cout << "Input sequence:\n";
  for(i=0; i<v.size(); i++) cout << v[i];
  cout << endl;

  // remove all spaces
  remove_copy(v.begin(), v.end(), v2.begin(), ' ');

  cout << "Result after removing spaces:\n";
  for(i=0; i<v2.size(); i++) cout << v2[i];
  cout << endl << endl;

  // **** now, demonstrate replace_copy ****
  cout << "Input sequence:\n";
  for(i=0; i<v.size(); i++) cout << v[i];
  cout << endl;

  // replace spaces with colons
  replace_copy(v.begin(), v.end(), v2.begin(), ' ', ':');

  cout << "Result after replacing spaces with colons:\n";
  for(i=0; i<v2.size(); i++) cout << v2[i];
  cout << endl << endl;

  return 0;
}
```

程序的输出如下所示：

```
Input sequence:
The STL is power programming.
Result after removing spaces:
TheSTLispowerprogramming.

Input sequence:
The STL is power programming.
Result after replacing spaces with colons:
The:STL:is:power:programming.
```

24.7.3 反转一个序列

一个经常使用的算法是 reverse( )，该算法可以反转一个序列，其一般形式如下：

```
template <class BiIter> void reverse(BiIter start, BiIter end);
```

reverse( )算法可以反转由 start 和 end 指定的范围内的元素顺序。

下面的程序演示了 reverse( )的使用方法。

```
// Demonstrate reverse.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

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

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

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

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

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

  return 0;
}
```

程序的输出如下所示：

```
Initial: 0 1 2 3 4 5 6 7 8 9
Reversed: 9 8 7 6 5 4 3 2 1 0
```

24.7.4 序列变换

还有一些更有趣的算法，其中之一是 transform( )，这个算法可以根据你提供的函数修改某个范围内的所有元素。transform( )算法具有以下两种通用形式：

```
template <class InIter, class OutIter, class Func)
  OutIter transform(InIter start, InIter end, OutIter result, Func unaryfunc);
```

```
template <class InIter1, class InIter2, class OutIter, class Func)
  OutIter transform(InIter1 start1, InIter1 end1, InIter2 start2,
                    OutIter result, Func binaryfunc);
```

transform( )算法把一个函数应用于某个范围的元素并把结果存储在 result 中。在第一种形式中，范围由 start 和 end 指定；被应用的函数由 unaryfunc 指定。该函数接收参数中的元素值并且返回相应的变换。在第二种形式中，使用一个二元运算符函数应用变换，该函数利用第一个参数接收一个序列中被变换的元素值，利用第二个参数接收另一个序列的元素。这两个版本都返回一个指向结果序列末端的迭代器。

下面的程序利用一个称为reciprocal( )的简单变换函数把一系列数字变换为它们的倒数。我们注意到，结果序列被存储到提供有原始序列的相同列表中。

```
// An example of the transform algorithm.
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

// A simple transformation function.
double reciprocal(double i) {
  return 1.0/i; // return reciprocal
}

int main()
{
  list<double> vals;
  int i;

  // put values into list
  for(i=1; i<10; i++) vals.push_back((double)i);

  cout << "Original contents of vals:\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  cout << endl;

  // transform vals
  p = transform(vals.begin(), vals.end(),
                vals.begin(), reciprocal);

  cout << "Transformed contents of vals:\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

这个程序的输出如下所示：

```
Original contents of vals:
1 2 3 4 5 6 7 8 9
Transformed contents of vals:
1 0.5 0.333333 0.25 0.2 0.166667 0.142857 0.125 0.111111
```

就像你看到的那样，vals 中的所有元素都被变换为它们的倒数。

24.8　使用函数对象

就像本章开始时讲到的那样，STL 支持（并广泛使用）函数对象。我们知道，函数对象是定义 operator( )的简单类。STL 提供许多内置函数对象，例如 less，minus 等。STL 还可以使你定义自己的函数对象。坦率地说，对有关函数对象的创建与使用的问题进行全面的讨论超出了本书的范围。幸运的是，我们可以像前面例子所示的那样使用 STL，而且不用创建函数对象。然而，由于函数对象是 STL 的主要成分，所以有必要对其进行一般性的了解。

24.8.1　一元与二元函数对象

就像有一元谓词和二元谓词一样，C++中也有一元函数对象和二元函数对象。一元函数对象需要一个参数，二元函数对象需要两个参数，而且在编程时必须使用要求的对象类型。例如，如果一个算法需要一个二元函数对象，则必须给它传递一个这种类型的对象。

24.8.2　使用内置函数对象

STL 提供了种类繁多的内置函数对象。下面列出了一些二元函数对象：

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

以下是一元函数对象：

logical_not	negate

函数对象执行的操作由它们的名字指定，惟一一个例外或许应该是 negate( )，该函数对象用来改变其参数的符号（正负号）。

内置函数对象是重载 operator( )的模板类，它返回对选定数据类型进行指定操作的结果。例如，为了对 float 数据调用二元函数对象 plus( )，可以采用下面的语法：

```
plus<float>()
```

内置函数对象需要用到头文件<functional>。

我们先来看一个简单的例子。下面的程序利用 transform( )算法（参见前面一节的描述）和 negate( )函数对象改变一系列数值的正负号。

```
// Use a unary function object.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
```

```
using namespace std;

int main()
{
  list<double> vals;
  int i;

  // put values into list
  for(i=1; i<10; i++) vals.push_back((double)i);

  cout << "Original contents of vals:\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  // use the negate function object
  p = transform(vals.begin(), vals.end(),
                vals.begin(),
                negate<double>()); // call function object

  cout << "Negated contents of vals:\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

程序的输出如下：

```
Original contents of vals:
1 2 3 4 5 6 7 8 9
Negated contents of vals:
-1 -2 -3 -4 -5 -6 -7 -8 -9
```

在这个程序中，我们可以注意到negate( )是如何被调用的。因为vals是一系列double值，所以程序利用negate<double>( )调用negate( )。针对序列中的每一个元素，transform( )算法都自动调用negate( )，因此，传给negate( )的单一参数将序列中的一个元素接收为它的变元。

下面的程序演示了二元函数对象divides( )的使用方法。该程序创建两个double值的列表并用一个列表中的元素除以另一个列表中的元素。这个程序使用了transform( )算法的二元形式。

```
// Use a binary function object.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
```

```
  list<double> vals;
  list<double> divisors;
  int i;

  // put values into list
  for(i=10; i<100; i+=10) vals.push_back((double)i);
  for(i=1; i<10; i++) divisors.push_back(3.0);

  cout << "Original contents of vals:\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  cout << endl;

  // transform vals
  p = transform(vals.begin(), vals.end(),
                divisors.begin(), vals.begin(),
                divides<double>()); // call function object

  cout << "Divided contents of vals:\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

程序的输出如下所示：

```
Original contents of vals:
10 20 30 40 50 60 70 80 90
Divided contents of vals:
3.33333 6.66667 10 13.3333 16.6667 20 23.3333 26.6667 30
```

在这个例子中，二元函数对象divides( )用第二个序列中的元素除第一个序列中的相应元素，因此，divides( )按照下面的指令接收参数：

divides(*first, second*)

上面的语句可以推而广之。只要使用二元函数对象，其参数的顺序将为 first, second。

24.8.3　创建函数对象

除了使用内置函数对象外，也可以创建自己的函数对象。为了达到这个目的，只需要创建一个重载 operator( )函数的对象。然而，为了获得最大的灵活性，你会希望使用下面这些类中的一个，这些类被 STL 定义为你所创建的函数对象的基类。

```
template <class Argument, class Result> struct unary_function {
  typedef Argument argument_type;
  typedef Result result_type;
```

```
};

template <class Argument1, class Argument2, class Result>
struct binary_function {
  typedef Argument1 first_argument_type;
  typedef Argument2 second_argument_type;
  typedef Result result_type;
};
```

这些模板类为函数对象使用的一般数据类型提供具体的类型名。虽然从技术上讲这些模板类很便利，但是它们几乎总是在创建函数对象时才被用到。

下面的程序演示了一个定制的函数对象，它将reciprocal()函数（前面用于演示transform()算法）转换为一个函数对象。

```
// Create a reciprocal function object.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

// A simple function object.
class reciprocal: unary_function<double, double> {
public:
  result_type operator()(argument_type i)
  {
    return (result_type) 1.0/i; // return reciprocal
  }
};

int main()
{
  list<double> vals;
  int i;

  // put values into list
  for(i=1; i<10; i++) vals.push_back((double)i);

  cout << "Original contents of vals:\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  // use reciprocal function object
  p = transform(vals.begin(), vals.end(),
                vals.begin(),
                reciprocal()); // call function object

  cout << "Transformed contents of vals:\n";
  p = vals.begin();
  while(p != vals.end()) {
```

```
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

我们注意到，reciprocal( )有两个重要方面。第一，它继承了基类unary_function，这使它可以访问 argument_type 和 result_type 类型。第二，它定义了 operator( )，从而可以返回其参数的倒数。一般来说，为了创建一个函数对象，只需继承适当的基类并根据需要重载 operator( )。

24.8.4　使用绑定器

当使用一个二元函数对象时，有可能把一个值绑定到某一个参数。这在许多情况下都很实用。例如，你或许想要从一个序列中删除所有大于某个值（例如 8）的元素。要达到这个目的，需要使用某种方法把 8 绑定到函数对象 greater( )右边的操作数。也就是说，想让 greater( )对序列中的每一个元素执行下面的比较：

```
val > 8
```

STL 提供了一种称为绑定器（binder）的机制，可以完成这个任务。

C++ 有两种绑定器：bind2nd( )和 bind1st( )，它们的一般形式如下：

```
bind1st(binfunc_obj, value)
bind2nd(binfunc_obj, value)
```

其中，binfunc_obj 是一个二元函数对象。bind1st( )返回一个一元函数对象，该函数对象具有绑定到 value 的 binfunc_obj 的左操作数；bind2nd( )返回一个一元函数对象，该函数对象具有绑定到 value 的 binfunc_obj 的右操作数。到目前为止，bind2nd( )是最常用的绑定器。无论是哪一种情况，一个绑定器产生的结果都是一个绑定到所指定值的一元函数对象。

为了说明绑定器的用途，我们将使用remove_if( )算法。该算法根据一个谓词结果删除一个序列中的元素，其原型如下：

```
template <class ForIter, class UnPred>
  ForIter remove_if(ForIter start, ForIter end, UnPred func);
```

如果 func 定义的一元谓词为 true，该算法从由 start 和 end 定义的序列中删除一些元素，然后返回一个指向新的序列末端的指针以反映出元素已被删除。

下面的程序从一个序列中删除所有大于 8 的元素。由于 remove_if( )要求一元谓词，所以不能简单地使用 greater( )函数对象，因为 greater( )是一个二元对象。相反，必须利用 bind2nd( )绑定器把数值 8 绑定到 greater( )的第二个参数，就像程序所示的那样。

```
// Demonstrate bind2nd().
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
```

```
{
  list<int> lst;
  list<int>::iterator p, endp;

  int i;

  for(i=1; i < 20; i++) lst.push_back(i);

  cout << "Original sequence:\n";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  endp = remove_if(lst.begin(), lst.end(),
                   bind2nd(greater<int>(), 8));

  cout << "Resulting sequence:\n";
  p = lst.begin();
  while(p != endp) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

程序的输出如下所示：

```
Original sequence:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Resulting sequence:
1 2 3 4 5 6 7 8
```

你或许想用这个程序做个试验，可以尝试不同的函数对象并绑定不同的值。你会发现，绑定器以非常有效的方式扩展了STL的功能。

最后一点：有一个与绑定器有关的对象，称为取反器（negator），取反器包括not1()和not2()。它们返回所修饰的谓词的非。取反器的一般形式如下：

not1(*unary_predicate*)
not2(*binary_predicate*)

例如，如果用下面的代码行代替前面程序中的相应语句，则将删除从lst开始的所有小于或等于8的元素。

```
endp = remove_if(lst.begin(), lst.end(),
                 not1(bind2nd(greater<int>(), 8)));
```

24.9 string类

我们已经知道，C++本身不支持内置字符串类型。然而，它却提供了两种处理字符串的方式。第一，可以利用传统的以Null结束的字符数组，读者对这种方法已经很熟悉了。这种字符

串有时被称为C字符串；第二，是作为string类型的一个类对象，这是我们要在这里讨论的方法。

实际上，string类是更具一般性的basic_string类的特例。事实上，basic_string有两个特例：一个是支持8位字符串的string，另一个是支持宽字符串的wstring。到目前为止，由于8位字符在一般编程中最常用，所以这里讨论string。

在介绍string类之前，有必要了解为什么将string类作为C++库的一部分。标准类不是被随便添加到C++的。事实上，针对每一个新添加的类，都伴随着各种各样的想法和争论。假定C++已经提供了一些对以Null结束的字符数组这样的字符串的支持，那么乍看上去，包含string类则是这个规则的一个例外情况。然而，这种看法是不对的，其原因是：以Null结束的字符串不能被任何标准C++运算符所操作，也不能作为一般的C++表达式的一部分。例如，看一看下面的代码片断：

```
char s1[ 80], s2[ 80], s3[ 80];

s1 = "Alpha"; // can't do
s2 = "Beta"; // can't do
s3 = s1 + s2; // error, not allowed
```

就像注释语句说明的那样，在C++中不可能利用赋值运算符给一个字符数组分配一个新值（初始化时除外），也不可能使用+运算符连接两个字符串。这些操作必须用库函数编写，如下所示：

```
strcpy(s1, "Alpha");
strcpy(s2, "Beta");
strcpy(s3, s1);
strcat(s3, s2);
```

因为从技术上讲，以Null结束的字符数组本身不是数据类型，所以不能对它们使用C++运算符，这使得最基本的字符串操作也显得笨拙。此外，也不能利用推动标准字符串类发展的标准C++运算符操作以Null结束的字符串。记住，当在C++中定义一个类时，实际上是在定义一个新的数据类型，该类型可以完全集成到C++环境中。当然，这意味着这些运算符可以被相关的新类重载。因此，通过添加一个标准的字符串类，有可能以操作其他数据类型的方法操纵字符串，即：利用运算符操作。

然而，使用标准字符串类还有另一个原因：出于安全的考虑。对于缺乏经验和粗心大意的程序员来说，包含以Null结束的字符串的数组非常容易发生数组越界的事情。例如，我们看一看标准的字符串拷贝函数strcpy()。这个函数不包含检查数组边界的规定。如果源数组包含的字符多于目标数组可以容纳的字符个数，那么可能出现程序错误或者系统崩溃的情况。你将会看到，标准string类可以防止出现这种错误。

最后，我们分析一下包括标准string类的3个原因，它们是一致性（现在，一个字符串定义一个数据类型）、方便性（可以使用标准C++运算符）和安全性（数组不会越界）。切记，没有任何理由可以完全放弃一般的以Null结束的字符串，它们仍然是实现字符串的最有效的方法。然而，当速度不是最重要的问题时，使用新的string类使你能够以一种全集成的安全方法操纵字符串。

虽然传统上我们不认为string是STL的一部分，但是它却是C++定义的另一个容器类。这

意味着它支持前面一节描述的算法。然而，字符串还有一些附加的功能。为了访问string类，必须在程序中包括<string>。

string类很大，它带有许多构造函数和成员函数，而且，许多成员函数具有多种重载形式。由于这个原因，我们不可能在本章介绍string的全部内容。相反，我们只介绍几个最常用的功能。一旦读者对string的工作方式有了一般性的了解，就可以很容易地深入探究其他内容。

string类支持几种构造函数，其中3个最常用的构造函数的原型如下：

```
string( );
string(const char *str);
string(const string &str);
```

第一种形式创建一个空的string对象；第二种形式从str所指的以Null结束的字符串中创建一个string对象，这种形式提供了一个从以Null结束的字符串到string对象的转换；第三种形式从另一个string中创建一个string。

C++为string对象定义了许多应用于字符串的运算符，其中包括：

运算符	含义
=	赋值
+	连接
+=	连接赋值
==	等于
!=	不等于
<	小于
<=	小于或等于
>	大于
>=	大于或等于
[]	下标
<<	输出
>>	输入

这些运算符允许在一般表达式中使用string对象并且不再需要调用诸如strcpy( )或strcat( )之类的函数。一般来说，可以在表达式中把string对象和一般的以Null结束的字符串混在一起使用。例如，可以把一个以Null结束的字符串赋给一个string对象。

+运算符可以用于把一个字符串对象和另一个字符串对象连接起来，或者把一个字符串对象和一个C字符串连接起来。也就是说，C++支持下列变体：

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

+运算符还可以用于把一个字符连接到一个字符串的末尾。string类定义了一个常量npos，它的值为-1，该常量表示有可能出现的最长的字符串长度。

C++的字符串类使得字符串的处理非常简单。例如，利用string对象，你可以用赋值运算符把一个引用字符串赋给一个string，用+运算符连接字符串，用比较运算符比较字符串。下面的程序举例说明了这些操作。

```
// A short string demonstration.
```

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

int main()
{
  string str1("Alpha");
  string str2("Beta");
  string str3("Omega");
  string str4;

  // assign a string
  str4 = str1;
  cout << str1 << "\n" << str3 << "\n";

  // concatenate two strings
  str4 = str1 + str2;
  cout << str4 << "\n";

  // concatenate a string with a C-string
  str4 = str1 + " to " + str3;
  cout << str4 << "\n";

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

  /* A string object can also be
     assigned a normal string. */
  str1 = "This is a null-terminated string.\n";
  cout << str1;

  // create a string object using another string object
  string str5(str1);
  cout << str5;

  // input a string
  cout << "Enter a string: ";
  cin >> str5;
  cout << str5;

  return 0;
}
```

这个程序的输出如下：

```
Alpha
Omega
AlphaBeta
Alpha to Omega
str3 > str1
This is a null-terminated string.
This is a null-terminated string.
Enter a string: STL
STL
```

从这个程序可以看出，操作字符串非常容易。例如，可以用+把字符串连接起来，可以用>对两个字符串进行比较。要使用C形式的以Null结束的字符串实现这些操作，要求调用strcat( )和strcmp( )函数。因为C++的string对象可以自由地与C形式的以Null结束的字符串混合在一起，所以在程序中使用它们不会有任何不良后果，相反，它们会为你带来很多益处。

在前面的程序中还有一件事情要注意：字符串的大小没有指定。string对象会自动调整大小以存放相应的字符串。因此，当分配或连接字符串时，目标字符串将根据需要扩展以便适应新的字符串长度，所以不可能出现字符串越界的情况。string对象这种动态特性是它们优于以Null结束的标准字符串（该字符串受限于边界范围）的诸多方面之一。

24.9.1 一些string成员函数

尽管利用字符串运算符可以实现最简单的操作，但是更复杂和精细的操作需要用string成员函数完成。虽然string有许多的成员函数需要讨论，但这里只能介绍几个最常用的成员函数。

基本字符串操作

为了把一个字符串赋给另一个字符串，可以利用assign( )函数。下面是该函数的两种形式：

```
string &assign(const string &strob, size_type start, size_type num);
string &assign(const char *str, size_type num);
```

在第一种形式中，函数将strob中从start指定的下标开始计数的num个字符赋给调用对象。在第二种形式中，以Null结束的字符串str的前num个字符被赋给调用对象。这两种形式的函数都返回一个对调用对象的引用。当然，使用该函数比使用运算符=把一个完整的字符串赋给另一个字符串要容易得多。只有当把部分字符串赋给另一个字符串时才必须使用assign( )函数。

可以利用append( )成员函数把某个字符串的一部分添加到另一个字符串的末端，该函数的两种形式如下所示：

```
string &append(const string &strob, size_type start, size_type num);
string &append(const char *str, size_type num);
```

在第一种形式中，函数将strob中从start指定的下标开始计数的num个字符添加到调用对象中。在第二种形式中，以Null结束的字符串str的前num个字符被添加到调用对象。这两种形式的函数都返回一个对调用对象的引用。当然，使用该函数比使用运算符+把一个完整的字符串添加到另一个字符串要容易得多。只有当把部分字符串添加到另一个字符串时才必须使用append( )函数。

可以利用insert( )和replace( )在一个字符串中插入或替换字符。这两个函数最常用的原型如下所示：

```
string &insert(size_type start, const string &strob);
string &insert(size_type start, const string &strob,
               size_type insStart, size_type num);
string &replace(size_type start, size_type num, const string &strob);
string &replace(size_type start, size_type orgNum, const string &strob,
                size_type replaceStart, size_type replaceNum);
```

第一种形式的insert( )函数按start指定的下标位置把strob插入到调用字符串中；第一种形式的insert( )函数按start指定的下标位置把strob中从insStart开始的num个字符插入到调用字

符串中。

第一种形式的 replace( )用 strob 替换调用字符串中从 start 开始的 num 个字符；第二种形式的 replace( )用 strob 指定的字符串中从 replaceStart 开始的 replaceNum 个字符替换调用字符串中从 start 开始的 orgNum 个字符。这两种形式的 replace( )函数都返回一个对调用对象的引用。

可以用 erase( )删除一个字符串中的字符，它的一种函数形式如下：

```
string &erase(size_type start = 0, size_type num = npos);
```

该函数从 start 开始删除调用字符串中的 num 个字符并返回一个对调用字符串的引用。

下面的程序演示了 insert( )，erase( )和 replace( )函数。

```
// Demonstrate insert(), erase(), and replace().
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1("String handling C++ style.");
  string str2("STL Power");

  cout << "Initial strings:\n";
  cout << "str1: " << str1 << endl;
  cout << "str2: " << str2 << "\n\n";

  // demonstrate insert()
  cout << "Insert str2 into str1:\n";
  str1.insert(6, str2);
  cout << str1 << "\n\n";

  // demonstrate erase()
  cout << "Remove 9 characters from str1:\n";
  str1.erase(6, 9);
  cout << str1 <<"\n\n";

  // demonstrate replace
  cout << "Replace 8 characters in str1 with str2:\n";
  str1.replace(7, 8, str2);
  cout << str1 << endl;

  return 0;
}
```

程序的输出如下所示：

```
Initial strings:
str1: String handling C++ style.
str2: STL Power

Insert str2 into str1:
StringSTL Power handling C++ style.

Remove 9 characters from str1:
String handling C++ style.

Replace 8 characters in str1 with str2:
```

```
String STL Power C++ style.
```

搜索字符串

string类提供了若干个搜索字符串的成员函数，包括find( )和rfind( )。下面是这些函数最常用的版本原型：

```
size_type find(const string &strob, size_type start=0) const;
size_type rfind(const string &strob, size_type start=npos) const;
```

find( )从start开始搜索调用字符串中第一次出现的包含在strob中的字符串。如果找到了，则返回调用字符串中出现匹配时的下标；如果没有找到，则返回npos。rfind( )与find( )相反，该函数从start开始反向搜索调用字符串中第一次出现的包含在strob中的字符串（即，搜索调用字符串中最后出现的strob）。如果找到了，rfind( )返回调用字符串中出现匹配时的下标；如果没有找到，则返回npos。

下面是一个使用find( )和rfind( )的简短例子。

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

int main()
{
  int i;
  string s1 =
    "Quick of Mind, Strong of Body, Pure of Heart";
  string s2;
  i = s1.find("Quick");
  if(i!=string::npos) {
    cout << "Match found at " << i << endl;
    cout << "Remaining string is:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
  cout << "\n\n";

  i = s1.find("Strong");
  if(i!=string::npos) {
    cout << "Match found at " << i << endl;
    cout << "Remaining string is:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
  cout << "\n\n";

  i = s1.find("Pure");
  if(i!=string::npos) {
    cout << "Match found at " << i << endl;
    cout << "Remaining string is:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
  cout << "\n\n";
```

```
  // find list "of"
  i = s1.rfind("of");
  if(i!=string::npos) {
    cout << "Match found at " << i << endl;
    cout << "Remaining string is:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }

  return 0;
}
```

程序的输出如下所示：

```
Match found at 0
Remaining string is:
Quick of Mind, Strong of Body, Pure of Heart

Match found at 15
Remaining string is:
Strong of Body, Pure of Heart

Match found at 31
Remaining string is:
Pure of Heart

Match found at 36
Remaining string is:
of Heart
```

比较字符串

为了把一个字符串对象的全部内容与另一个字符串对象做比较，通常可以使用本书前面介绍的重载关系运算符。然而，如果想对某字符串的一部分内容与另一个字符串进行比较，则需使用下面的compare( )成员函数：

int compare(size_type *start*, size_type *num*, const string &*strob*) const;

其中，函数将对strob中从start开始的num个字符与调用字符串进行比较。如果调用字符串小于strob，compare( )的返回值小于0；如果调用字符串大于strob，函数的返回值大于0；如果调用字符串等于strob，函数的返回值为0。

获取以Null结束的字符串

虽然string对象很实用，但是有时也需要获取一个以Null结束的字符串数组。例如，你有可能使用一个string对象构造一个文件名。然而，当打开文件时，需要指定一个指向以Null结束的标准字符串的指针。为了解决这个问题，C++提供了一个称为c_str( )的成员函数。该函数的原型如下：

const char *c_str( ) const;

这个函数返回一个指向包含在调用string对象中的以Null结束的字符串的指针。这个以Null结束的字符串不能被修改，而且在对该string对象进行了其他操作之后也不能保证其有效性。

24.9.2 字符串就是容器

string类符合一个容器的所有基本要求，它支持常用的容器函数，例如begin( )，end( )和size( )。此外，它还支持迭代器，因此，也可以利用STL算法操作string对象。下面是一个简单的例子：

```
// Strings as containers.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
  string str1("Strings handling is easy in C++");
  string::iterator p;
  unsigned int i;

  // use size()
  for(i=0; i<str1.size(); i++)
    cout << str1[i];
  cout << endl;

  // use iterator
  p = str1.begin();
  while(p != str1.end())
    cout << *p++;
  cout << endl;

  // use the count() algorithm
  i = count(str1.begin(), str1.end(), 'i');
  cout << "There are " << i << " i's in str1\n";

  // use transform() to upper case the string
  transform(str1.begin(), str1.end(), str1.begin(),
            toupper);
  p = str1.begin();
  while(p != str1.end())
    cout << *p++;
  cout << endl;

  return 0;
}
```

程序的输出如下所示：

```
Strings handling is easy in C++
Strings handling is easy in C++
There are 4 i's in str1
STRINGS HANDLING IS EASY IN C++
```

24.9.3 将字符串放入其他容器中

虽然string是一个容器，但是字符串类型的对象通常存放在其他STL容器中，例如映射容器或列表容器。下面的程序对于编写前面所示的电话号码簿来说是一种更好的方法。该程序使

用一个 string 对象的映射（而不是以 Null 结束的字符串）存放人名和电话号码。

```
// Use a map of strings to create a phone directory.
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  map<string, string> directory;

  directory.insert(pair<string, string>("Tom", "555-4533"));
  directory.insert(pair<string, string>("Chris", "555-9678"));
  directory.insert(pair<string, string>("John", "555-8195"));
  directory.insert(pair<string, string>("Rachel", "555-0809"));

  string s;
  cout << "Enter name: ";
  cin >> s;

  map<string, string>::iterator p;

  p = directory.find(s);
  if(p != directory.end())
    cout << "Phone number: " << p->second;
  else
    cout << "Name not in directory.\n";

  return 0;
}
```

24.10　关于 STL 的最后一点说明

STL 是 C++ 语言非常重要的组成部分，许多编程工作都可以根据 STL 来设计完成。STL 将功能性和灵活性集于一体，虽然语法有些复杂，但是它非常易用。没有一个 C++ 的编程人员能够忽视 STL，因为它将在未来的编程中扮演重要角色。