第33章 STL容器类

本章描述实现由标准模板库（STL）定义的容器的类。容器是标准模板库的部件，它提供了对其他对象的存储。除了存储对象所需的内存外，它们也定义了访问容器中的对象的机制。因此，容器是高级存储设备。

注意：关于STL的概述和教程，请参见第24章。

在容器说明中，我们会看到下面的约定。当涉及各种通用的迭代器类型时，本书将使用这里所列的术语。

术语	表示
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

当要求一元谓词函数时，使用UnPred来表示。当要求二元谓词时，使用BinPred。在二元谓词中，参数总是按相对于调用谓词的函数为第一、第二的顺序出现。对既有一元谓词又有二元谓词的情况，参数将包含被容器存储的对象类型的值。

比较函数使用Comp来表示。

还有，在以后的描述中，当我们说迭代器指向容器的末尾时，这意味着迭代器指向容器中最后一个对象的后面。

33.1 容器类

由STL定义的容器如下表所示。

容器	说明	所要求的头文件
bitset	位的集合	<bitset>
deque	以双精度数结束的队列	<deque>
list	线性表	<list>
map	存储键/值对，其中每个键仅与一个值相关联	<map>
multimap	存储键/值对，其中一个键可以与两个或更多的值相关联	<map>
multiset	一个集合，其中每个元素不必是惟一的	<set>
priority_queue	优先级队列	<queue>
queue	一个队列	<queue>
set	一个集合，其中每个元素是惟一的	<set>
stack	一个堆栈	<stack>
vector	一个动态数组	<vector>

下面几节总结了每个容器。因为容器是使用模板类来实现的，所以使用了各种占位符数据类型。在说明中，通用类型T表示容器所存的数据类型。因为模板类中占位符类型的名称是任意的，容器类声明了这些类型的typedef版本，这使得类型名更具体了。下面是容器类所用的typedef名称。

size_type	大致等于size_t的某些整数类型
reference	一个元素的引用
const_reference	一个元素的const引用
difference_type	可以表示两个地址间的差值
iterator	迭代器
const_iterator	const迭代器
reverse_iterator	反向迭代器
const_reverse_iterator	const反向迭代器
value_type	容器中存储的值的类型（通常与通用类型T相同）
allocator_type	分配器的类型
key_type	键的类型
key_compare	比较两个键的函数的类型
mapped_type	存储在一个映射中的值的类型（与通用类型T相同）
value_compare	比较两个值的函数的类型
pointer	指针类型
const_pointer	const指针类型
container_type	容器的类型

33.1.1 bitset

bitset类支持对位的集合进行操作，它的模板规范是：

```
template <size_t N> class bitset;
```

其中，N规定了位集的长度（以位计）。它的构造函数如下：

```
bitset( );
bitset(unsigned long bits);
explicit bitset(const string &s, size_t i = 0, size_t num = npos);
```

第一种形式构造一个空的位集。第二种形式构造一个位集，该位集一致于在bits中指定的那些位集。第三种形式使用在i处开始的字符串s构造一个位集，字符串必须仅包含1和0。只使用num或s.size( )-i中小的那个。常量npos是一个大得足以说明s的最大长度的值。

输出运算符<<和>>是为bitset定义的。

bitset包含下面的成员函数。

成员	说明
bool any( ) const;	如果在调用位集中的任何位是1，返回真。否则，返回假
size_t count( ) const;	返回1位的数目
bitset<N> &flip( );	颠倒在调用位集中所有位的状态并返回*this
bitset<N> &flip(size_t *i*);	颠倒在调用位集中位置i处的位并返回*this
bool none( ) const;	如果在调用位集中没有设置位，返回真
bool operator !=(const bitset<N> &*op2*) const;	如果调用位集不同于右边运算符op2所指定的位集，返回真

（续表）

成员	说明
bool operator ==(const bitset<N> &*op2*) const;	如果调用位集与右边运算符 op2 所指定的位集相同，返回真
bitset<N>&operator &=(const bitset<N> &*op2*);	把调用位集中的每一位和 op2 中的对应位相“与”，并把结果放到调用位集中。它返回 *this
bitset<N>&operator ^=(const bitset<N> &*op2*);	把调用位集中的每一位和 op2 中的对应位相“异或”，并把结果放到调用位集中。它返回 *this
bitset<N>&operator |=(const bitset<N> &*op2*);	把调用位集中的每一位和 op2 中的对应位相“或”，并把结果放到调用位集中。它返回 *this
bitset<N> &operator ~( ) const;	颠倒调用位集中所有位的状态并返回结果
bitset<N> &operator <<=(size_t *num*);	左移调用位集中的每一位 num 个位置，并把结果放到调用位集中。它返回 *this
bitset<N> &operator >>=(size_t *num*);	右移调用位集中的每一位 num 个位置，并把结果放到调用位集中。它返回 *this
reference operator [ ](size_t *i*);	返回到调用位集中位 i 的一个引用
bitset<N> &reset( );	清除调用位集中的所有位并返回 *this
bitset<N> &reset(size_t *i*);	清除调用位集中位置 i 处的位并返回 *this
bitset<N> &set( );	设置调用位集中的所有位并返回 *this
bitset<N> &set(size_t *i*, int *val* = 1);	设置位置 i 处的位为调用位集中由 val 指定的值并返回 *this。val 的任何非 0 值被假定为 1
size_t size( ) const;	返回位集所含的位数
bool test(size_t *i*) const;	返回位置 i 处位的状态
string to_string( ) const;	返回调用位集中包含位模式表示的一个字符串
unsigned long to_ulong( ) const;	把调用位集转换为无符号长整数

33.1.2　deque

deque 类支持双端队列，它的模板规范是：

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

其中，T 是在 deque 中存储的数据类型。它具有下面的构造函数：

```
explicit deque(const Allocator &a = Allocator( ) );

explicit deque(size_type num, const T &val = T ( ),
  const Allocator &a = Allocator( ));

deque(const deque<T, Allocator> &ob);

template <class InIter> deque(InIter start, InIter end,
  const Allocator &a = Allocator( ));
```

第一种形式构造一个空双端队列。第二种形式构造一个双端队列，该队列具有 num 个其值为 val 的元素。第三种形式构造一个双端队列，该队列包含与 ob 一样多的元素。第四种形式构造一个双端队列，该队列包含由 start 和 end 所指定的范围的元素。

下面的比较运算符是为 deque 定义的：

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

deque包含下表列出的成员函数。

成员	说明
template <class InIter> void assign(InIter *start*, InIter *end*);	把由start和end定义的序列赋给这个双端队列
void assign(size_type *num*, const T &*val*);	把num个值为val的元素赋给这个双端队列
reference at(size_type *i*); const_reference at(size_type *i*) const;	返回一个到由i所指的元素的引用
reference back( ); const_reference back( ) const;	返回一个到双端队列中最后一个元素的引用
iterator begin( ); const_iterator begin( ) const;	返回指向双端队列中第一个元素的迭代器
void clear( );	删除双端队列中的所有元素
bool empty( ) const:	如果调用的双端队列为空，返回真。否则，返回假
const_iterator end( ) const; iterator end( );	返回指向双端队列末尾的迭代器
iterator erase(iterator *i*);	删除i所指的元素。返回指向所删除元素后面元素的迭代器
iterator erase(iterator *start*, iterator *end*);	删除从start到end这一范围的元素。返回指向所删除的最后一个元素后面元素的迭代器
reference front( ); const_reference front( ) const;	返回到双端队列中第一个元素的引用
allocator_type get_allocator( ) const;	返回双端队列的分配器
iterator insert(iterator *i*, const T &*val*);	把val插入到由i所指的元素的前一个元素。返回一个指向该元素的迭代器
void insert(iterator *i*, size_type *num*, const T &*val*);	把val的num个副本插入到由i所指的元素的前一个元素处
template <class InIter> void insert(iterator *i*, InIter start, InIter *end*);	把由start和end所定义的序列插入到由i所指的元素的前一个元素处
size_type max_size( ) const;	返回该双端队列所能包含的元素的最大数目
reference operator[ ](size_type *i*); const_reference operator[ ](size_type *i*) const;	返回到第i个元素的引用
void pop_back( );	删除双端队列中的最后一个元素
void pop_front( );	删除双端队列中的第一个元素
void push_back(const T &*val*);	把具有由val所指定的值的一个元素加到双端队列的末尾
void push_front(const T &*val*);	把具有由val所指定的值的一个元素加到双端队列的开始处
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	返回一个指向双端队列末尾的反向迭代器
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	返回一个指向双端队列开始处的反向迭代器
void resize(size_type *num*, T *val* = T ( ));	改变双端队列大小为由num指定的那个大小。如果必须加长此双端队列，那么具有由val所指定的值的元素被加到末尾
size_type size( ) const;	返回当前双端队列中的元素数
void swap(deque<T, Allocator> &*ob*);	把存储在调用双端队列中的元素和ob中的元素交换

33.1.3 list

list 类支持列表，它的模板规范是：

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

其中，T 是存储在列表中的数据类型。它具有下面的构造函数：

```
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，该列表具有 num 个值为 val 的元素。第三种形式构造一个列表，该列表包含与 ob 相同的元素。第四种形式构造一个列表，该列表包含由 start 和 end 所指定的范围中的元素。

下面的比较运算符是为列表定义的：

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

list 包含下表列出的成员函数。

成员	说明
template <class InIter> void assign(InIter *start*, InIter *end*);	把由 start 和 end 定义的序列赋给列表
void assign(size_type *num*, const T &*val*);	把值为 val 的 num 个元素赋给列表
reference back( ); const_reference back( ) const;	返回到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，返回真。否则，返回假
iterator end( ); const_iterator end( ) const;	返回指向列表末尾的迭代器
iterator erase(iterator *i*);	删除i所指的元素。返回指向所删除元素后面元素的迭代器
iterator erase(iterator *start*, iterator *end*);	除去在从 start 到 end 范围内的元素。返回指向所删除的最后一个元素后面元素的迭代器
reference front( ); const_reference front( ) const;	返回到列表中第一个元素的引用
allocator_type get_allocator( ) const;	返回列表的分配器
iterator insert(iterator *i*, const T &*val* = T( ));	把 val 插到由 i 所指定的元素的前一个元素处。返回指向那个元素的迭代器
void insert(iterator *i*, size_type *num*, const T & *val*);	把值为 val 的 num 个副本插到由 i 所指定的元素的前一个元素处

（续表）

成员	说明
template <class InIter> void insert(iterator *i*,InIter *start*, InIter *end*);	把由start和end定义的序列插到由i所指的元素的前一个元素处
size_type max_size( ) const;	返回列表中所能包含的元素的最大数目
void merge(list<T, Allocator> &*ob*); template <class Comp> void merge(<list<T, Allocator> &*ob*, Comp *cmpfn*);	把包含于ob中的有序列表和有序调用列表合并。结果为有序表。在合并后，包含于ob中的列表为空。在第二种形式中，可以指定一个比较函数，此函数决定了何时一个元素比另一个元素小
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*);	把具有由val所指定值的一个元素加到列表的末尾
void push_front(const T &*val*);	把具有由val所指定值的一个元素加到列表的前面
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	返回一个指向列表末尾的反向迭代器
void remove(const T &*val*);	把值为val的元素从列表中删除
template <class UnPred> void remove_if(UnPred *pr*);	删除使一元谓词pr为真的元素
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	删除指向列表开始处的一个反向迭代器
void resize(size_type *num*, T *val* = T ( ));	改变列表的大小为num指定的大小。如果必须加长列表，那么把其值为由val指定的值的元素加到末尾
void reverse( );	颠倒调用列表
size_type size( ) const;	返回当前在列表中的元素数

void sort( );
template <class Comp>
void sort(Comp *cmpfn*);
	排序列表。第二种形式使用比较函数cmpfn来排序列表，以决定何时一个元素比另一个元素小

void splice(iterator *i*, list<T, Allocator> &*ob*);	把ob的内容插入到调用列表中i所指的位置。完成此操作后，ob为空
void splice(iterator *i*, list<T, Allocator> &*ob*, iterator *el*);	把el所指的元素从列表ob中删除并存储在调用列表中i所指的位置
void splice(iterator *i*, list<T, Allocator> &*ob*, iterator *start*, iterator *end*);	把start和end所定义的范围从ob中去除并存储在以i所指的位置开始的调用列表中
void swap(list<T, Allocator> &*ob*);	把存储在调用列表中的元素和ob中的元素相交换

void unique( );
template <class BinPred>
void unique(BinPred *pr*);
	删除调用列表中相同的元素。第二种形式使用pr来决定惟一性

33.1.4 map

map类支持一个关联容器，其中惟一键和值相互映射。它的模板规范如下所示：

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

其中，Key是键的数据类型，T是所存储的值的数据类型（映射的），Comp是一个比较两

个键的函数。它有下面的构造函数：

```
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包含与ob同样的元素的映射。第三种形式构造一个映射，该映射包含由 start 和 end 所指定范围中的元素。由 comfn 指定的函数，如果存在，决定了映射的顺序。

下面的比较运算符是为 map 定义的。

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

map 所包含的成员函数如下表所示。在说明中，key_type 是键的类型，value_type 表示 pair<Key, T>。

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;	返回在映射中 k 出现的次数（1 或 0）
bool empty( ) const;	如果调用映射为空，返回真。否则，返回假
iterator end( ); const_iterator end( ) const;	返回一个指向映射末尾的迭代器

pair<iterator, iterator>
equal_range(const key_type &k);
pair<const_iterator, const_iterator>
equal_range(const key_type &k) const;
	返回指向包含指定键的映射的第一个元素和最后一个元素的一对迭代器

void erase(iterator i);	删除 i 所指的元素
void erase(iterator start, iterator end);	删除从 start 开始、到 end 结束这一范围的元素
size_type erase(const key_type &k);	从映射中删除其键值为 k 的元素
iterator find(const key_type &k); const_iterator find(const key_type &k) const;	返回指向所指定键的迭代器。如果没有找到键，则返回指向映射末尾的迭代器
allocator_type get_allocator( ) const;	返回映射的分配器
iterator insert(iterator i, const value_type &val);	在i所指的元素处或其后插入val。返回指向该元素的迭代器
template <class InIter> void insert(InIter start, InIter end);	插入某一范围的元素

（续表）

成员	说明
pair<iterator, bool> insert(const value_type &*val*);	把val插入到调用映射中。返回指向这个元素的迭代器。如果这个元素还不存在，才插入这个元素。如果已插入了这个元素，返回pair<iterator, true>。否则，返回pair<iterator, false>
key_compare key_comp( ) const;	返回比较键的函数对象

iterator lower_bound(const key_type &*k*);
const_iterator
lower_bound(const key_type &*k*) const;
	返回指向映射中其键等于或大于k的第一个元素的迭代器

size_type max_size( ) const;	返回映射所能容纳的元素的最大数目
mapped_type & operator[ ] (const key_type &*i*);	返回到i所指的元素的引用。如果这个元素不存在，那么插入它
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	返回指向映射末尾的反向迭代器
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	返回指向映射开始处的反向迭代器
size_type size( ) const;	返回当前在映射中元素的数目
void swap(map<Key, T, Comp, Allocator> &*ob*);	把存储在调用映射中的元素和在ob中的元素交换

iterator upper_bound(const key_type &*k*);
const_iterator
upper_bound(const key_type &k) const;
	返回指向映射中键大于k的第一个元素的迭代器

value_compare value_comp( ) const;	返回比较值的函数对象

33.1.5 multimap

multimap类支持一个关联容器，在这个容器中，可能有非惟一值与值相映射。它的模板规范如下所示：

```
template <class Key, class T, class Comp = less<Key>,
          class Allocator = allocator<pair<const Key, T > > > class multimap
```

其中，Key是键的数据类型，T是所存储（所映射的）值的数据类型，Comp是比较两个键的函数。它有下面的构造函数：

```
explicit multimap(const Comp &cmpfn = Comp( ),
const Allocator &a = Allocator( ) );

multimap(const multimap<Key, T, Comp, Allocator> &ob);

template <class InIter> multimap(InIter start, InIter end,
            const Comp &cmpfn = Comp( ),
            const Allocator &a = Allocator( ));
```

第一种形式构造一个空的多映射。第二种形式构造一个包含与ob相同的元素的多映射。第三种形式构造一个多映射，该多映射包含由start和end所指定的范围中的元素。由cmpfn指定的函数，如果存在的话，决定了多映射的顺序。

下面的比较运算符由multimap所定义：

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

multimap 所包含的成员函数如下表所示。在说明中，key_type 是键的类型，T 是值，value_type 表示 pair<Key, T>。

成员	说明
iterator begin( ); const_iterator begin( ) const;	返回指向多映射中第一个元素的迭代器
void clear( );	从多映射中删除所有的元素
size_type count(const key_type &*k*) const;	返回多映射中 k 出现的次数
bool empty( ) const;	如果调用多映射为空，返回真。否则，返回假
iterator end( ); const_iterator end( ) const;	返回指向列表末尾的迭代器

pair<iterator, iterator>
equal_range(const key_type &*k*);
pair<const_iterator, const_iterator>
equal_range(const key_type &*k*) const;
	返回指向多映射中包含指定键的第一个元素和最后一个元素的一对迭代器

void erase(iterator *i*);	删除 i 所指的元素
void erase(iterator *start*, iterator *end*);	删除在从 start 到 end 的范围中的元素
size_type erase(const key_type &*k*);	从多映射中删除其键值为 k 的元素
iterator find(const key_type &*k*); const_iterator find(const key_type &*k*) const;	返回指向所指定键的迭代器。如果没有找到这个键，那么返回指向多映射末尾的迭代器
allocator_type get_allocator( ) const;	返回多映射的分配器
iterator insert(iterator *i*, const value_type &*val*);	把 val 插到 i 所指的元素处或其后。返回指向该元素的迭代器
template <class InIter> void insert(InIter *start*, InIter *end*);	插入某一范围的元素
iterator insert(const value_type &*val*);	把 val 插到调用的多映射中
key_compare key_comp( ) const;	返回比较键的函数对象

iterator lower_bound(const key_type &*k*);
const_iterator
lower_bound(const key_type &*k*) const;
	返回指向多映射中其键等于或大于 k 的第一个元素的迭代器

size_type max_size( ) const;	返回多映射所能容纳的元素的最大数目
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	返回指向多映射末尾的反向迭代器
reverse_iterator rend( );	返回指向多映射开始处的反向迭代器
const_reverse_iterator rend( ) const;	
size_type size( ) const;	返回当前在多映射中的元素数
void swap(multimap<Key, T, Comp, Allocator> &*ob*);	把存储在调用多映射中的元素和 ob 中的元素相交换

iterator upper_bound(const key_type &*k*);
const_iterator
upper_bound(const key_type &*k*) const;
	返回指向多映射中其键大于 k 的第一个元素的迭代器

value_compare value_comp( ) const;	返回比较值的函数对象

33.1.6 multiset

multiset类支持可能包含非惟一键的一个集合，它的模板规范如下所示：

```
template <class Key, class Comp = less<Key>,
          class Allocator = allocator<Key> > class multiset
```

其中，Key是键的数据类型，Comp是比较两个键的函数。它有下面的构造函数：

```
explicit multiset(const Comp &cmpfn = Comp( ),
                  const Allocator &a = Allocator( ) );

multiset(const multiset<Key, Comp, Allocator> &ob);

template <class InIter> multiset(InIter start, InIter end,
                  const Comp &cmpfn = Comp( ),
                  const Allocator &a = Allocator( ));
```

第一种形式构造一个空的multiset。第二种形式构造一个multiset，该multiset包含与ob同样的元素。第三种形式构造一个multiset，该multiset包含在start和end所指定范围内的元素。由cmpfn所规定的函数，如果存在的话，决定集合的顺序。

下面的比较运算符是为multiset定义的。

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

multiset所包含的成员函数如下表所示。在说明中，key_type和value_type都是Key的typedef。

成员	说明
iterator begin( ); const_iterator begin( ) const;	返回指向multiset中第一个元素的迭代器
void clear( );	删除multiset中的所有元素
size_type count(const key_type &*k*) const;	返回在multiset中k出现的次数
bool empty( ) const;	如果调用multiset为空，返回真。否则，返回假
iterator end( ); const_iterator end( ) const;	返回指向multiset末尾的迭代器
pair<iterator, iterator> equal_range(const key_type &*k*) const;	返回指向包含指定键的multiset中第一个和最后一个元素的一对迭代器
void erase(iterator *i*);	删除i所指的元素
void erase(iterator *start*, iterator *end*);	删除在从start到end的范围中的元素
size_type erase(const key_type &*k*);	删除multiset中其键值为k的元素
iterator find(const key_type &*k*) const;	返回指向所给定键的迭代器。如果没有找到键，那么返回指向multiset末尾的迭代器
allocator_type get_allocator( ) const;	返回multiset的分配器
iterator insert(iterator *i*, const value_type &*val*);	把val插到i所指的元素处或该元素的后面。返回指向该元素的迭代器
template <class InIter> void insert(InIter *start*, InIter *end*);	插入某一范围的元素

（续表）

成员	说明
iterator insert(const value_type &*val*);	把 val 插到调用 multiset 中。返回指向那个元素的迭代器
key_compare key_comp( ) const;	返回比较键的函数对象
iterator lower_bound(const key_type &*k*) const;	返回指向 multiset 中其键等于或大于 k 的第一个元素的迭代器
size_type max_size( ) const;	返回 multiset 所能容纳的元素的最大数目
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	返回指向 multiset 末尾的反向迭代器
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	返回指向 multiset 开始处的反向迭代器
size_type size( ) const;	返回当前在 multiset 中元素的数目
void swap(multiset<Key, Comp, Allocator> &*ob*);	把存储在调用 multiset 中的元素和 ob 中的元素相交换
iterator upper_bound(const key_type &*k*) const;	返回指向 multiset 中其键大于 k 的第一个元素的迭代器
value_compare value_comp( ) const;	返回比较值的函数对象

33.1.7　queue

queue 类支持单端队列，它的模板规范如下所示：

```
template <class T, class Container = deque<T> > class queue
```

其中，T 是所存储的数据类型，Container 是用于容纳队列的容器类型。它有下面的构造函数：

```
explicit queue(const Container &cnt = Container( ));
```

queue( )构造函数创建一个空队列。默认时，它使用一个 deque 作为容器，但是 queue 仅可通过先进、先出方式来访问。容器包含于一个 Container 的受保护对象 c 中。

下面的比较运算符是为 queue 定义的：

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

queue 包含下表列出的成员函数。

成员	说明
value_type &back( ); const vulue_type &back( ) const;	返回到队列中最后一个元素的引用
bool empty( ) const;	如果调用队列为空，返回真。否则，返回假
value_type &front( ); const value_type &front( ) const;	返回到队列中第一个元素的引用
void pop( );	删除队列中的第一个元素
void push(const value_type &*val*);	把其值由 val 指定的一个元素加到队列的末尾
size_type size( ) const;	返回当前在队列中的元素数

33.1.8　priority_queue

priority_queue 类支持单端优先级队列，它的模板规范如下所示：

```
template <class T, class Container = vector<T>,
          class Comp = less<Container::value_type> >
          class priority_queue
```

其中，T是所存储的数据类型，Container是用于容纳队列的容器类型，Comp指定比较函数，该函数决定何时优先级队列的一个成员在优先级上比另一个成员低。它有下面的构造函数：

```
explicit priority_queue(const Comp &cmpfn = Comp( ),
          Container &cnt = Container( ));

template <class InIter> priority_queue(InIter start, InIter end,
          const Comp &cmpfn = Comp( ),
          Container &cnt = Container( ));
```

第一种priority_queue( )构造函数创建一个空的优先级队列。第二种创建一个优先级队列，该队列包含由start和end所指定的范围内的元素。默认时，它使用一个vector作为容器。也可以使用deque作为优先级队列的容器，这个容器包含于Container的一个受保护对象c中。priority_queue包含下表列出的成员函数。

成员	说明
bool empty( ) const;	如果调用优先级队列为空，返回真。否则，返回假
void pop( );	删除优先级队列中的第一个元素
void push(const T &val);	把一个元素加到优先级队列中
size_type size( ) const;	返回在当前优先级队列中的元素数
const value_type &top( ) const;	返回到具有最高优先级的元素的引用。不能删除这个元素

33.1.9 set

set类支持包含惟一键的一个集合，它的模板规范如下所示：

```
template <class Key, class Comp = less<Key>,
          class Allocator = allocator<Key> > class set
```

其中，Key是键的数据，Comp是比较两个键的函数。它有下面的构造函数：

```
explicit set(const Comp &cmpfn = Comp( ),
          const Allocator &a = Allocator( ) );

set(const set<Key, Comp, Allocator> &ob);

template <class InIter> set(InIter start, InIter end,
          const Comp &cmpfn = Comp( ),
          const Allocator &a = Allocator( ));
```

第一种形式构造一个空集合。第二种形式构造一个集合，该集合包含与ob同样的元素。第三种形式构造一个集合，该集合包含由start和end所指定的范围内的元素。由cmpfn所指定的函数，如果存在的话，决定了集合的顺序。

下面的比较运算符是为set定义的：

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

set所包含的成员函数如下表所示。

成员	说明
iterator begin( ); const_iterator begin( ) const;	返回指向集合中第一个元素的迭代器
void clear( );	删除集合中的所有元素
size_type count(const key_type &*k*) const;	返回集合中 k 出现的次数
bool empty( ) const;	如果调用集合为空，返回真。否则，返回假
const_iterator end( ) const; iterator end( );	返回指向集合末尾的迭代器
pair<iterator, iterator> equal_range(const key_type &*k*) const;	返回指向包含指定键的集合中第一个元素和最后一个元素的一对迭代器
void erase(iterator *i*);	删除 i 所指的元素
void erase(iterator *start*, iterator *end*);	删除从 start 到 end 这一范围内的元素
size_type erase(const key_type &*k*);	删除集合中其键值为 k 的元素。返回所删除的元素数
iterator find(const key_type &*k*) const;	返回指向所指定键的迭代器。如果没有找到键，那么返回指向集合末尾的迭代器
allocator_type get_allocator( ) const;	返回集合的分配器
iterator insert(iterator *i*, const value_type &*val*);	把 val 插到 i 所指元素处或其后，不要插入相同的元素。返回指向该元素的迭代器
template <class InIter> void insert(InIter *start*, InIter *end*);	插入某一范围的元素。不要插入相同的元素
pair<iterator, bool> insert(const value_type &*val*);	把 val 插到调用集合中。返回指向该元素的迭代器。仅在元素不存在时才插入它。如果已插入了这个元素，返回 pair <iterator, true>。否则，返回 pair<iterator, false>
iterator lower_bound(const key_type &*k*) const;	返回指向集合中其键等于或大于 k 的第一个元素的迭代器
key_compare key_comp( ) const;	返回比较键的函数对象
size_type max_size( ) const;	返回集合所能容纳的元素的最大数目
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	返回指向集合末尾的反向迭代器
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	返回指向集合开始处的反向迭代器
size_type size( ) const;	返回当前在集合中的元素数
void swap(set<Key, Comp,Allocator> &*ob*);	把存储在调用集合中的元素和 ob 中的元素相交换
iterator upper_bound(const key_type &*k*) const;	返回指向集合中其键大于 k 的第一个元素的迭代器
value_compare value_comp( ) const;	返回比较值的函数对象

33.1.10　stack

stack 类支持堆栈，它的模板规范如下所示：

```
template <class T, class Container = deque<T> > class stack
```

其中，T 是所存储的数据类型，Container 是用来容纳堆栈的容器类型。它有下面的构造函数：

```
explicit stack(const Container &cnt = Container( ));
```

stack( )构造函数创建一个空堆栈。默认时，它使用一个deque作为容器，但是stack只能以后进、先出的方式访问。也可以使用一个vector或list作为堆栈的容器。容器包含于Container的受保护成员c中。

下面的比较运算符是为stack定义的：

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

stack包含下表列出的成员函数。

成员	说明
bool empty( ) const;	如果调用堆栈为空，返回真。否则，返回假
void pop( );	删除堆栈顶部，技术上这是容器中的最后一个元素
void push(const value_type &*val*);	把一个元素压入堆栈中。容器中的最后一个元素表示堆栈顶部
size_type size( ) const;	返回当前在堆栈中的元素数
value_type &top( );	返回到堆栈顶部的引用，这是容器中的最后一个元素。不能删除这个元素
cont value_type &top( ) const;	

33.1.11 vector

vector类支持动态数组，它的模板规范如下所示：

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

其中，T是所存储的数据类型，Allocator指定分配器。它有下面的构造函数：

```
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矢量，该矢量具有num个其值为val的元素。第三种形式构造一个矢量，该矢量包含与ob同样的元素。第四种形式构造一个矢量，该矢量包含由start和end指定的范围中的元素。

下面的比较运算符是为vector定义的：

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

vector包含下表列出的成员函数。

成员	说明
template <class InIter> void assign(InIter *start*, InIter *end*);	把由start和end定义的序列赋给矢量
void assign(size_type *num*, const T &*val*);	把值为val的num个元素赋给矢量
reference at(size_type *i*); const_reference at(size_type *i*) const;	返回到由i所指的元素的引用
reference back( ); const_reference back( ) const;	返回到矢量中最后一个元素的引用

（续表）

成员	说明
iterator begin( ); const_iterator begin( ) const;	返回指向矢量中第一个元素的迭代器
size_type capacity( ) const;	返回矢量的当前容量。这是在它需要分配更多的内存前可以容纳的元素数
void clear( );	删除矢量中的所有元素
bool empty( ) const;	如果调用矢量为空，返回真。否则，返回假
iterator end( ); const_iterator end( ) const;	返回指向矢量末尾的迭代器
iterator erase(iterator *i*);	删除i所指的元素。返回指向所删除元素后面元素的迭代器
iterator erase(iterator *start*, iterator *end*);	删除在start到end范围内的元素。返回指向所删除的最后一个元素后面那个元素的迭代器
reference front( ); const_reference front( ) const;	返回到矢量中第一个元素的引用
allocator_type get_allocator( ) const;	返回矢量的分配器
iterator insert(iterator *i*, const T &*val*);	把 val 插入到 i 所指的元素之前的元素。返回指向该元素的迭代器
void insert(iterator *i*, size_type *num*, const T & *val*);	把 val 的 num 个副本插入到 i 所指的元素前面的元素处
template <class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	把 start 和 end 定义的序列插入到 i 所指的元素的前面
size_type max_size( ) const;	返回矢量能够容纳的元素的最大数目
reference operator[ ](size_type *i*) const; const_reference operator[ ](size_type *i*) const;	返回到 i 所指的元素的引用
void pop_back( );	删除矢量中的最后一个元素
void push_back(const T &*val*);	把具有 val 所指定的值的一个元素加到矢量的末尾
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	返回指向矢量末尾的反向迭代器
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	返回指向矢量开始处的反向迭代器
void reserve(size_type *num*);	设置矢量的容量，以便它至少等于 num
void resize(size_type *num*, T val = T ( ));	改变矢量的大小为num所指定的大小。如果必须加长矢量，那么把其值由 val 指定的元素加到末尾
size_type size( ) const;	返回当前在矢量中的元素数
void swap(vector<T, Allocator> &*ob*);	把存储在调用矢量中的元素和 ob 中的元素相交换

STL 也包含一个布尔值的 vector 规范。它包括所有的 vector 功能，并且添加了下面两个成员：

void flip( );	颠倒矢量中的所有位
static void swap(reference *i*, reference *j*);	交换 i 和 j 所指定的位