Chapter 33

The STL Container Classes

This chapter describes the classes that implement the containers defined by the standard template library (STL). Containers are the part of the STL that provide storage for other objects. In addition to supplying the memory necessary to store objects, they define the mechanisms by which the objects in the container may be accessed. Thus, containers are high-level storage devices.

Note *For an overview and tutorial to the STL, refer to Chapter 24.*

In the container descriptions, the following conventions will be observed. When referring to the various iterator types generically, this book will use the terms listed here.

Term	Represents
BiIter	Bidirectional iterator
ForIter	Forward iterator
InIter	Input iterator
OutIter	Output iterator
RandIter	Random access iterator

When a unary predicate function is required, it will be notated using the type **UnPred**. When a binary predicate is required, the type **BinPred** will be used. In a binary predicate, the arguments are always in the order of *first,second* relative to the function that calls the predicate. For both unary and binary predicates, the arguments will contain values of the type of objects being stored by the container.

Comparison functions will be notated using the type **Comp**.

One other point: In the descriptions that follow, when an iterator is said to point to the end of a container, this means that the iterator points just beyond the last object in the container.

The Container Classes

The containers defined by the STL are shown here.

Container	Description	Required Header
bitset	A set of bits.	<bitset>
deque	A double-ended queue.	<deque>
list	A linear list.	<list>

Container	**Description**	**Required Header**
map	Stores key/value pairs in which each key is associated with only one value.	<map>
multimap	Stores key/value pairs in which one key may be associated with two or more values.	<map>
multiset	A set in which each element is not necessarily unique.	<set>
priority_queue	A priority queue.	<queue>
queue	A queue.	<queue>
set	A set in which each element is unique.	<set>
stack	A stack.	<stack>
vector	A dynamic array.	<vector>

Each of the containers is summarized in the following sections. Since the containers are implemented using template classes, various placeholder data types are used. In the descriptions, the generic type **T** represents the type of data stored by a container.

Since the names of the placeholder types in a template class are arbitrary, the container classes declare **typedef**ed versions of these types. This makes the type names concrete. Here are the **typedef** names used by the container classes.

size_type	Some integral type roughly equivalent to **size_t**.
reference	A reference to an element.
const_reference	A **const** reference to an element.
difference_type	Can represent the difference between two addresses.
iterator	An iterator.
const_iterator	A **const** iterator.
reverse_iterator	A reverse iterator.
const_reverse_iterator	A **const** reverse iterator.

value_type	The type of a value stored in a container. (Often the same as the generic type T.)
allocator_type	The type of the allocator.
key_type	The type of a key.
key_compare	The type of a function that compares two keys.
mapped_type	The type of value stored in a map. (Same as the generic type T.)
value_compare	The type of a function that compares two values.
pointer	The type of a pointer.
const_pointer	The type of a **const** pointer.
container_type	The type of a container.

bitset

The **bitset** class supports operations on a set of bits. Its template specification is

template <size_t *N*> class bitset;

Here, *N* specifies the length of the bitset, in bits. It has the following constructors:

bitset( );

bitset(unsigned long *bits*);

explicit bitset(const string &*s*, size_t *i* = 0, size_t *num* = npos);

The first form constructs an empty bitset. The second form constructs a bitset that has its bits set according to those specified in *bits*. The third form constructs a bitset using the string *s*, beginning at *i*. The string must contain only 1's and 0's. Only *num* or *s*.**size**( )-*i* values are used, whichever is less. The constant **npos** is a value that is sufficiently large to describe the maximum length of *s*.

The output operators << and >> are defined for **bitset**.

bitset contains the following member functions.

Member	**Description**
bool any( ) const;	Returns true if any bit in the invoking bitset is 1; otherwise returns false.
size_t count( ) const;	Returns the number of 1 bits.
bitset<N> &flip( );	Reverses the state of all bits in the invoking bitset and returns ***this.**
bitset<N> &flip(size_t *i*);	Reverses the bit in position *i* in the invoking bitset and returns ***this**.
bool none( ) const;	Returns true if no bits are set in the invoking bitset.
bool operator !=(const bitset<N> &*op2*) const;	Returns true if the invoking bitset differs from the one specified by right-hand operator, *op2*.
bool operator ==(const bitset<N> &*op2*) const;	Returns true if the invoking bitset is the same as the one specified by right-hand operator, *op2*.
bitset<N> &operator &=(const bitset<N> &*op2*);	ANDs each bit in the invoking bitset with the corresponding bit in *op2* and leaves the result in the invoking bitset. It returns ***this**.
bitset<N> &operator ^=(const bitset<N> &*op2*);	XORs each bit in the invoking bitset with the corresponding bit in *op2* and leaves the result in the invoking bitset. It returns ***this**.
bitset<N> &operator |=(const bitset<N> &*op2*);	ORs each bit in the invoking bitset with the corresponding bit in *op2* and leaves the result in the invoking bitset. It returns ***this**.
bitset<N> &operator ~( ) const;	Reverses the state of all bits in the invoking bitset and returns the result.
bitset<N> &operator <<=(size_t *num*);	Left-shifts each bit in the invoking bitset *num* positions and leaves the result in the invoking bitset. It returns ***this**.
bitset<N> &operator >>=(size_t *num*);	Right-shifts each bit in the invoking bitset *num* positions and leaves the result in the invoking bitset. It returns ***this**.

Member	Description
reference operator [ ](size_t *i*);	Returns a reference to bit *i* in the invoking bitset.
bitset<N> &reset( );	Clears all bits in the invoking bitset and returns ***this**.
bitset<N> &reset(size_t *i*);	Clears the bit in position *i* in the invoking bitset and returns ***this**.
bitset<N> &set( );	Sets all bits in the invoking bitset and returns ***this**.
bitset<N> &set(size_t *i*, int *val* = 1);	Sets the bit in position *i* to the value specified by *val* in the invoking bitset and returns ***this**. Any nonzero value for *val* is assumed to be 1.
size_t size( ) const;	Returns the number of bits that the bitset can hold.
bool test(size_t *i*) const;	Returns the state of the bit in position *i*.
string to_string( ) const;	Returns a string that contains a representation of the bit pattern in the invoking bitset.
unsigned long to_ulong( ) const;	Converts the invoking bitset into an unsigned long integer.

deque

The **deque** class supports a double-ended queue. Its template specification is

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

Here, **T** is the type of data stored in the **deque**. It has the following constructors:

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( ));

The first form constructs an empty deque. The second form constructs a deque that has *num* elements with the value *val*. The third form constructs a deque that contains the same elements as *ob*. The fourth form constructs a deque that contains the elements in the range specified by *start* and *end*.

The following comparison operators are defined for **deque**:

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

deque contains the following member functions.

Member	**Description**
template <class InIter> void assign(InIter *start*, InIter *end*);	Assigns the deque the sequence defined by *start* and *end*.
void assign(size_type *num*, const T &*val*);	Assigns the deque *num* elements of value *val*.

reference at(size_type *i*);
const_reference at(size_type *i*) const;
	Returns a reference to the element specified by *i*.

reference back( );
const_reference back( ) const;
	Returns a reference to the last element in the deque.

iterator begin( );
const_iterator begin( ) const;
	Returns an iterator to the first element in the deque.

void clear( );	Removes all elements from the deque.
bool empty( ) const;	Returns true if the invoking deque is empty and false otherwise.

const_iterator end( ) const;
iterator end( );
	Returns an iterator to the end of the deque.

iterator erase(iterator *i*);	Removes the element pointed to by *i*. Returns an iterator to the element after the one removed.
iterator erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*. Returns an iterator to the element after the last element removed.

reference front( );
const_reference front( ) const;
	Returns a reference to the first element in the deque.

Member	Description
allocator_type get_allocator( ) const;	Returns deque's allocator.
iterator insert(iterator *i*, const T &*val*);	Inserts *val* immediately before the element specified by *i*. An iterator to the element is returned.
void insert(iterator *i*, size_type *num*, const T &*val*);	Inserts *num* copies of *val* immediately before the element specified by *i*.
template <class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	Inserts the sequence defined by *start* and *end* immediately before the element specified by *i*.
size_type max_size( ) const;	Returns the maximum number of elements that the deque can hold.
reference operator[ ](size_type *i*); const_reference operator[ ](size_type *i*) const;	Returns a reference to the *i*th element.
void pop_back( );	Removes the last element in the deque.
void pop_front( );	Removes the first element in the deque.
void push_back(const T &*val*);	Adds an element with the value specified by *val* to the end of the deque.
void push_front(const T &*val*);	Adds an element with the value specified by *val* to the front of the deque.
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	Returns a reverse iterator to the end of the deque.
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	Returns a reverse iterator to the start of the deque.
void resize(size_type *num*, T *val* = T ( ));	Changes the size of the deque to that specified by *num*. If the deque must be lengthened, then elements with the value specified by *val* are added to the end.
size_type size( ) const;	Returns the number of elements currently in the deque.
void swap(deque<T, Allocator> &*ob*);	Exchanges the elements stored in the invoking deque with those in *ob*.

list

The **list** class supports a list. Its template specification is

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

Here, **T** is the type of data stored in the list. It has the following constructors:

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( ));

The first form constructs an empty list. The second form constructs a list that has *num* elements with the value *val*. The third form constructs a list that contains the same elements as *ob*. The fourth form constructs a list that contains the elements in the range specified by *start* and *end*.

The following comparison operators are defined for **list**:

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

list contains the following member functions.

Member	**Description**
template <class InIter> void assign(InIter *start*, InIter *end*);	Assigns the list the sequence defined by *start* and *end*.
void assign(size_type *num*, const T &*val*);	Assigns the list *num* elements of value *val*.
reference back( ); const_reference back( ) const;	Returns a reference to the last element in the list.
iterator begin( ); const_iterator begin( ) const;	Returns an iterator to the first element in the list.

Member	Description
void clear( );	Removes all elements from the list.
bool empty( ) const;	Returns true if the invoking list is empty and false otherwise.

iterator end( );
const_iterator end( ) const;
	Returns an iterator to the end of the list.

iterator erase(iterator *i*);	Removes the element pointed to by *i*. Returns an iterator to the element after the one removed.
iterator erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*. Returns an iterator to the element after the last element removed.

reference front( );
const_reference front( ) const;
	Returns a reference to the first element in the list.

allocator_type get_allocator( ) const;	Returns list's allocator.
iterator insert(iterator *i*, const T &*val* = T( ));	Inserts *val* immediately before the element specified by *i*. An iterator to the element is returned.
void insert(iterator *i*, size_type *num*, const T & *val*);	Inserts *num* copies of *val* immediately before the element specified by *i*.
template <class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	Inserts the sequence defined by *start* and *end* immediately before the element specified by *i*.
size_type max_size( ) const;	Returns the maximum number of elements that the list can hold.

void merge(list<T, Allocator> &*ob*);
template <class Comp>
void merge(<list<T, Allocator> &*ob*,
Comp *cmpfn*);
	Merges the ordered list contained in *ob* with the ordered invoking list. The result is ordered. After the merge, the list contained in *ob* is empty. In the second form, a comparison function can be specified that determines when one element is less than another.

void pop_back( );	Removes the last element in the list.
void pop_front( );	Removes the first element in the list.

Member	Description
void push_back(const T &*val*);	Adds an element with the value specified by *val* to the end of the list.
void push_front(const T &*val*);	Adds an element with the value specified by *val* to the front of the list.

reverse_iterator rbegin( );
const_reverse_iterator rbegin( ) const;
	Returns a reverse iterator to the end of the list.

void remove(const T &*val*);	Removes elements with the value *val* from the list.
template <class UnPred> void remove_if(UnPred *pr*);	Removes elements for which the unary predicate *pr* is true.

reverse_iterator rend( );
const_reverse_iterator rend( ) const;
	Returns a reverse iterator to the start of the list.

void resize(size_type *num*, T *val* = T ( ));	Changes the size of the list to that specified by *num*. If the list must be lengthened, then elements with the value specified by *val* are added to the end.
void reverse( );	Reverses the invoking list.
size_type size( ) const;	Returns the number of elements currently in the list.

void sort( );
template <class Comp>
void sort(Comp *cmpfn*);
	Sorts the list. The second form sorts the list using the comparison function *cmpfn* to determine when one element is less than another.

void splice(iterator *i*, list<T, Allocator> &*ob*);	The contents of *ob* are inserted into the invoking list at the location pointed to by *i*. After the operation, *ob* is empty.
void splice(iterator *i*, list<T, Allocator> &*ob*, iterator *el*);	The element pointed to by *el* is removed from the list *ob* and stored in the invoking list at the location pointed to by *i*.
void splice(iterator *i*, list<T, Allocator> &ob, iterator *start*, iterator *end*);	The range defined by *start* and *end* is removed from *ob* and stored in the invoking list beginning at the location pointed to by *i*.

Member	Description
void swap(list<T, Allocator> &*ob*);	Exchanges the elements stored in the invoking list with those in *ob*.

void unique( );
template <class BinPred>
  void unique(BinPred *pr*);
	Removes duplicate elements from the invoking list. The second form uses *pr* to determine uniqueness.

map

The **map** class supports an associative container in which unique keys are mapped with values. Its template specification is shown here:

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

Here, **Key** is the data type of the keys, **T** is the data type of the values being stored (mapped), and **Comp** is a function that compares two keys. It has the following constructors:

```
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( ));
```

The first form constructs an empty map. The second form constructs a map that contains the same elements as *ob*. The third form constructs a map that contains the elements in the range specified by *start* and *end*. The function specified by *cmpfn*, if present, determines the ordering of the map.

The following comparison operators are defined for **map**.

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

The member functions contained by **map** are shown here. In the descriptions, **key_type** is the type of the key, and **value_type** represents **pair<Key, T>**.

Member	**Description**

iterator begin( );
const_iterator begin( ) const;
	Returns an iterator to the first element in the map.

void clear( );	Removes all elements from the map.
size_type count(const key_type &*k*) const;	Returns the number of times *k* occurs in the map (1 or zero).
bool empty( ) const;	Returns true if the invoking map is empty and false otherwise.

iterator end( );
const_iterator end( ) const;
	Returns an iterator to the end of the map.

pair<iterator, iterator>
  equal_range(const key_type &*k*);
pair<const_iterator, const_iterator>
  equal_range(const key_type &*k*) const;
	Returns a pair of iterators that point to the first and last elements in the map that contain the specified key.

void erase(iterator *i*);	Removes the element pointed to by *i*.
void erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*.
size_type erase(const key_type &*k*);	Removes from the map elements that have keys with the value *k*.

iterator find(const key_type &*k*);
const_iterator find(const key_type &*k*)
  const;
	Returns an iterator to the specified key. If the key is not found, then an iterator to the end of the map is returned.

allocator_type get_allocator( ) const;	Returns map's allocator.
iterator insert(iterator *i*, const value_type &*val*);	Inserts *val* at or after the element specified by *i*. An iterator to the element is returned.
template <class InIter> void insert(InIter *start*, InIter *end*);	Inserts a range of elements.
pair<iterator, bool> insert(const value_type &*val*);	Inserts *val* into the invoking map. An iterator to the element is returned. The element is only inserted if it does not already exist. If the element was inserted, **pair<iterator, true>** is returned. Otherwise, **pair<iterator, false>** is returned.

Member	Description
key_compare key_comp( ) const;	Returns the function object that compares keys.

iterator lower_bound(const key_type &*k*);
const_iterator
lower_bound(const key_type &*k*) const;
	Returns an iterator to the first element in the map with the key equal to or greater than *k*.

size_type max_size( ) const;	Returns the maximum number of elements that the map can hold.
mapped_type & operator[ ] (const key_type &*i*);	Returns a reference to the element specified by *i*. If this element does not exist, it is inserted.

reverse_iterator rbegin( );
const_reverse_iterator rbegin( ) const;
	Returns a reverse iterator to the end of the map.

reverse_iterator rend( );
const_reverse_iterator rend( ) const;
	Returns a reverse iterator to the start of the map.

size_type size( ) const;	Returns the number of elements currently in the map.
void swap(map<Key, T, Comp, Allocator> &*ob*);	Exchanges the elements stored in the invoking map with those in *ob*.

iterator upper_bound(const key_type &*k*);
const_iterator
upper_bound(const key_type &k) const;
	Returns an iterator to the first element in the map with the key greater than *k*.

value_compare value_comp( ) const;	Returns the function object that compares values.

multimap

The **multimap** class supports an associative container in which possibly nonunique keys are mapped with values. Its template specification is shown here:

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

Here, **Key** is the data type of the keys, **T** is the data type of the values being stored (mapped), and **Comp** is a function that compares two keys. It has the following constructors:

```
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( ));
```

The first form constructs an empty multimap. The second form constructs a multimap that contains the same elements as *ob*. The third form constructs a multimap that contains the elements in the range specified by *start* and *end*. The function specified by *cmpfn*, if present, determines the ordering of the multimap.

The following comparison operators are defined by **multimap**:

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

The member functions contained by **multimap** are shown here. In the descriptions, **key_type** is the type of the key, **T** is the value, and **value_type** represents **pair<Key, T>**.

Member	**Description**

iterator begin( );
const_iterator begin( ) const;
	Returns an iterator to the first element in the multimap.

void clear( );	Removes all elements from the multimap.
size_type count(const key_type &*k*) const;	Returns the number of times *k* occurs in the multimap.
bool empty( ) const;	Returns true if the invoking multimap is empty and false otherwise.

iterator end( );
const_iterator end( ) const;
	Returns an iterator to the end of the list.

pair<iterator, iterator>
equal_range(const key_type &*k*);
pair<const_iterator, const_iterator>
equal_range(const key_type &*k*) const;
	Returns a pair of iterators that point to the first and last elements in the multimap that contain the specified key.

void erase(iterator *i*);	Removes the element pointed to by *i*.
void erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*.

Member	**Description**
size_type erase(const key_type &*k*);	Removes from the multimap elements that have keys with the value *k*.

iterator find(const key_type &*k*);
const_iterator find(const key_type &*k*) const;
	Returns an iterator to the specified key. If the key is not found, then an iterator to the end of the multimap is returned.

allocator_type get_allocator( ) const;	Returns multimap's allocator.
iterator insert(iterator *i*, const value_type &*val*);	Inserts *val* at or after the element specified by *i*. An iterator to the element is returned.
template <class InIter> void insert(InIter *start*, InIter *end*);	Inserts a range of elements.
iterator insert(const value_type &*val*);	Inserts *val* into the invoking multimap.
key_compare key_comp( ) const;	Returns the function object that compares keys.

iterator lower_bound(const key_type &*k*);
const_iterator lower_bound(const key_type &*k*) const;
	Returns an iterator to the first element in the multimap with the key equal to or greater than *k*.

size_type max_size( ) const;	Returns the maximum number of elements that the multimap can hold.

reverse_iterator rbegin( );
const_reverse_iterator rbegin( ) const;
	Returns a reverse iterator to the end of the multimap.

reverse_iterator rend( );
const_reverse_iterator rend( ) const;
	Returns a reverse iterator to the start of the multimap.

size_type size( ) const;	Returns the number of elements currently in the multimap.
void swap(multimap<Key, T, Comp, Allocator> &*ob*);	Exchanges the elements stored in the invoking multimap with those in *ob*.

iterator upper_bound(const key_type &*k*);
const_iterator upper_bound(const key_type &*k*) const;
	Returns an iterator to the first element in the multimap with the key greater than *k*.

value_compare value_comp( ) const;	Returns the function object that compares values.

multiset

The **multiset** class supports a set containing possibly nonunique keys. Its template specification is shown here:

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

Here, **Key** is the data of the keys and **Comp** is a function that compares two keys. It has the following constructors:

```
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( ));
```

The first form constructs an empty multiset. The second form constructs a multiset that contains the same elements as *ob*. The third form constructs a multiset that contains the elements in the range specified by *start* and *end*. The function specified by *cmpfn*, if present, determines the ordering of the set.

The following comparison operators are defined for **multiset**.

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

The member functions contained by **multiset** are shown here. In the descriptions, both **key_type** and **value_type** are **typedef**s for **Key**.

Member	**Description**

iterator begin( );
const_iterator begin( ) const;
	Returns an iterator to the first element in the multiset.

void clear( );	Removes all elements from the multiset.
size_type count(const key_type &*k*) const;	Returns the number of times *k* occurs in the multiset.
bool empty( ) const;	Returns true if the invoking multiset is empty and false otherwise.

Member	Description

iterator end( );
const_iterator end( ) const;
	Returns an iterator to the end of the multiset.

pair<iterator, iterator> equal_range(const key_type &*k*) const;	Returns a pair of iterators that point to the first and last elements in the multiset that contain the specified key.
void erase(iterator *i*);	Removes the element pointed to by *i*.
void erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*.
size_type erase(const key_type &*k*);	Removes from the multiset elements that have keys with the value *k*.
iterator find(const key_type &*k*) const;	Returns an iterator to the specified key. If the key is not found, then an iterator to the end of the multiset is returned.
allocator_type get_allocator( ) const;	Returns multiset's allocator.
iterator insert(iterator *i*, const value_type &*val*);	Inserts *val* at or after the element specified by *i*. An iterator to the element is returned.
template <class InIter> void insert(InIter *start*, InIter *end*);	Inserts a range of elements.
iterator insert(const value_type &*val*);	Inserts *val* into the invoking multiset. An iterator to the element is returned.
key_compare key_comp( ) const;	Returns the function object that compares keys.
iterator lower_bound(const key_type &*k*) const;	Returns an iterator to the first element in the multiset with the key equal to or greater than *k*.
size_type max_size( ) const;	Returns the maximum number of elements that the multiset can hold.

reverse_iterator rbegin( );
const_reverse_iterator rbegin( ) const;
	Returns a reverse iterator to the end of the multiset.

reverse_iterator rend( );
const_reverse_iterator rend( ) const;
	Returns a reverse iterator to the start of the multiset.

Member	Description
size_type size( ) const;	Returns the number of elements currently in the multiset.
void swap(multiset<Key, Comp, Allocator> &*ob*);	Exchanges the elements stored in the invoking multiset with those in *ob*.
iterator upper_bound(const key_type &*k*) const;	Returns an iterator to the first element in the multiset with the key greater than *k*.
value_compare value_comp( ) const;	Returns the function object that compares values.

queue

The **queue** class supports a single-ended queue. Its template specification is shown here:

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

Here, **T** is the type of data being stored and **Container** is the type of container used to hold the queue. It has the following constructor:

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

The **queue( )** constructor creates an empty queue. By default it uses a **deque** as a container, but a **queue** can only be accessed in a first-in, first-out manner. You can also use a **list** as a container for a queue. The container is held in a protected object called **c** of type **Container**.

The following comparison operators are defined for **queue**:

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

queue contains the following member functions.

Member	Description

value_type &back( );
const value_type &back( ) const;
	Returns a reference to the last element in the queue.

bool empty( ) const;	Returns true if the invoking queue is empty and false otherwise.

Member	Description

value_type &front( );
const value_type &front( ) const;
	Returns a reference to the first element in the queue.

void pop( );	Removes the first element in the queue.
void push(const value_type &*val*);	Adds an element with the value specified by *val* to the end of the queue.
size_type size( ) const;	Returns the number of elements currently in the queue.

priority_queue

The **priority_queue** class supports a single-ended priority queue. Its template specification is shown here:

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

Here, **T** is the type of data being stored. **Container** is the type of container used to hold the queue, and **Comp** specifies the comparison function that determines when one member for the priority queue is lower in priority than another. It has the following constructors:

```
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( ));
```

The first **priority_queue( )** constructor creates an empty priority queue. The second creates a priority queue that contains the elements specified by the range *start* and *end*. By default it uses a **vector** as a container. You can also use a **deque** as a container for a priority queue. The container is held in a protected object called **c** of type **Container**.

priority_queue contains the following member functions.

Member	**Description**
bool empty( ) const;	Returns true if the invoking priority queue is empty and false otherwise.
void pop( );	Removes the first element in the priority queue.
void push(const T &*val*);	Adds an element to the priority queue.
size_type size( ) const;	Returns the number of elements current in the priority queue.
const value_type &top( ) const;	Returns a reference to the element with the highest priority. The element is not removed.

set

The **set** class supports a set containing unique keys. Its template specification is shown here:

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

Here, **Key** is the data of the keys and **Comp** is a function that compares two keys. It has the following constructors:

```
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( ));
```

The first form constructs an empty set. The second form constructs a set that contains the same elements as *ob*. The third form constructs a set that contains the elements in the range specified by *start* and *end*. The function specified by *cmpfn*, if present, determines the ordering of the set.

The following comparison operators are defined for **set**:

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

The member functions contained by **set** are shown here.

Member	**Description**

iterator begin( );
const_iterator begin( ) const;
	Returns an iterator to the first element in the set.

void clear( );	Removes all elements from the set.
size_type count(const key_type &*k*) const;	Returns the number of times *k* occurs in the set.
bool empty( ) const;	Returns true if the invoking set is empty and false otherwise.

const_iterator end( ) const;
iterator end( );
	Returns an iterator to the end of the set.

pair<iterator, iterator> equal_range(const key_type &*k*) const;	Returns a pair of iterators that point to the first and last elements in the set that contain the specified key.
void erase(iterator *i*);	Removes the element pointed to by *i*.
void erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*.
size_type erase(const key_type &*k*);	Removes from the set elements that have keys with the value *k*. The number of elements removed is returned.
iterator find(const key_type &*k*) const;	Returns an iterator to the specified key. If the key is not found, then an iterator to the end of the set is returned.
allocator_type get_allocator( ) const;	Returns set's allocator.
iterator insert(iterator *i*, const value_type &*val*);	Inserts *val* at or after the element specified by *i*. Duplicate elements are not inserted. An iterator to the element is returned.
template <class InIter> void insert(InIter *start*, InIter *end*);	Inserts a range of elements. Duplicate elements are not inserted.

Member	**Description**
pair<iterator, bool> insert(const value_type &*val*);	Inserts *val* into the invoking set. An iterator to the element is returned. The element is inserted only if it does not already exist. If the element was inserted, **pair<iterator, true>** is returned. Otherwise, **pair<iterator, false>** is returned.
iterator lower_bound(const key_type &*k*) const;	Returns an iterator to the first element in the set with the key equal to or greater than *k*.
key_compare key_comp( ) const;	Returns the function object that compares keys.
size_type max_size( ) const;	Returns the maximum number of elements that the set can hold.
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	Returns a reverse iterator to the end of the set.
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	Returns a reverse iterator to the start of the set.
size_type size( ) const;	Returns the number of elements currently in the set.
void swap(set<Key, Comp,Allocator> &*ob*);	Exchanges the elements stored in the invoking set with those in *ob*.
iterator upper_bound(const key_type &*k*) const;	Returns an iterator to the first element in the set with the key greater than *k*.
value_compare value_comp( ) const;	Returns the function object that compares values.

stack

The **stack** class supports a stack. Its template specification is shown here:

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

Here, **T** is the type of data being stored and **Container** is the type of container used to hold the stack. It has the following constructor:

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

The **stack( )** constructor creates an empty stack. By default it uses a **deque** as a container, but a **stack** can only be accessed in a last-in, first-out manner. You may also use a **vector** or **list** as a container for a stack. The container is held in a protected member called **c** of type **Container**.

The following comparison operators are defined for **stack**:

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

stack contains the following member functions.

Member	**Description**
bool empty( ) const;	Returns true if the invoking stack is empty and false otherwise.
void pop( );	Removes the top of the stack, which is technically the last element in the container.
void push(const value_type &*val*);	Pushes an element onto the end of the stack. The last element in the container represents the top of the stack.
size_type size( ) const;	Returns the number of elements currently in the stack.

value_type &top( );
cont value_type &top( ) const;
	Returns a reference to the top of the stack, which is the last element in the container. The element is not removed.

vector

The **vector** class supports a dynamic array. Its template specification is shown here.

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

Here, **T** is the type of data being stored and **Allocator** specifies the allocator. It has the following constructors.

```
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( ));
```

The first form constructs an empty vector. The second form constructs a vector that has *num* elements with the value *val*. The third form constructs a vector that contains the same elements as *ob*. The fourth form constructs a vector that contains the elements in the range specified by *start* and *end*.

The following comparison operators are defined for **vector**:

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

vector contains the following member functions.

Member	**Description**
template <class InIter> void assign(InIter *start*, InIter *end*);	Assigns the vector the sequence defined by *start* and *end*.
void assign(size_type *num*, const T &*val*);	Assigns the vector *num* elements of value *val*.

reference at(size_type *i*);
const_reference at(size_type *i*) const;
	Returns a reference to an element specified by *i*.

reference back( );
const_reference back( ) const;
	Returns a reference to the last element in the vector.

iterator begin( );
const_iterator begin( ) const;
	Returns an iterator to the first element in the vector.

size_type capacity( ) const;	Returns the current capacity of the vector. This is the number of elements it can hold before it will need to allocate more memory.
void clear( );	Removes all elements from the vector.
bool empty( ) const;	Returns true if the invoking vector is empty and false otherwise.

Member	Description

iterator end( );
const_iterator end( ) const;
	Returns an iterator to the end of the vector.

iterator erase(iterator *i*);	Removes the element pointed to by *i*. Returns an iterator to the element after the one removed.
iterator erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*. Returns an iterator to the element after the last element removed.

reference front( );
const_reference front( ) const;
	Returns a reference to the first element in the vector.

allocator_type get_allocator( ) const;	Returns vector's allocator.
iterator insert(iterator *i*, const T &*val*);	Inserts *val* immediately before the element specified by *i*. An iterator to the element is returned.
void insert(iterator *i*, size_type *num*, const T & *val*);	Inserts *num* copies of *val* immediately before the element specified by *i*.
template <class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	Inserts the sequence defined by *start* and *end* immediately before the element specified by *i*.
size_type max_size( ) const;	Returns the maximum number of elements that the vector can hold.

reference operator[ ](size_type *i*) const;
const_reference operator[ ](size_type *i*)
const;
	Returns a reference to the element specified by *i*.

void pop_back( );	Removes the last element in the vector.
void push_back(const T &*val*);	Adds an element with the value specified by *val* to the end of the vector.

reverse_iterator rbegin( );
const_reverse_iterator rbegin( ) const;
	Returns a reverse iterator to the end of the vector.

reverse_iterator rend( );
const_reverse_iterator rend( ) const;
	Returns a reverse iterator to the start of the vector.

Member	**Description**
void reserve(size_type *num*);	Sets the capacity of the vector so that it is equal to at least *num*.
void resize(size_type *num*, T val = T ( ));	Changes the size of the vector to that specified by *num*. If the vector must be lengthened, then elements with the value specified by *val* are added to the end.
size_type size( ) const;	Returns the number of elements currently in the vector.
void swap(vector<T, Allocator> &*ob*);	Exchanges the elements stored in the invoking vector with those in *ob*.

The STL also contains a specialization of **vector** for Boolean values. It includes all of the functionality of **vector** and adds these two members.

void flip( );	Reverses all bits in the vector.
static void swap(reference *i*, reference *j*);	Exchanges the bits specified by *i* and *j*.