BC2

Standard Library Reference

This reference shows the most useful classes and functions in the standard library.

Note that the syntax "[start, end)" refers to a half-open iterator range from start to end, as described in Chapter 21.

Containers

The STL divides its containers into three categories. The sequential containers include the `vector`, `list`, and `deque`. The associative containers include the `map`, `multimap`, `set`, and `multiset`. The container adapters include the `stack`, `queue`, and `priority_queue`. Additionally, the `bitmap` and `string` can be considered STL containers as well.

Common Typedefs

Most sequential and associative containers that are part of the standard define the following types, which have public access and are used in the method prototypes. Note that not all these `typedefs` are required by the standard in order to qualify as an STL container. However, most of the containers in the STL provide them (exceptions are noted after the following table).

Type Name	Description
`value_type`	The element type stored in the container.
`reference`	A reference to the element type stored in the container.
`const_reference`	A reference to a `const` element type stored in the container.
`pointer`	A pointer to the element type with which the container is instantiated (not required by the standard, but defined by all the containers).
`const_pointer`	A pointer to a `const` element type with which the container is instantiated (not required by the standard, but defined by all the containers).
`iterator`	The type of the "smart pointer" for iterating over elements of the container.
`const_iterator`	A version of `iterator` for iterating over `const` elements of the container.
`reverse_iterator`	The type of the "smart pointer" for iterating over elements of the container in reverse order.
`const_reverse_iterator`	A version of `reverse_iterator` for iterating over `const` elements of the container.
`size_type`	Type that can represent the number of elements in the container. Usually just `size_t` (from `<cstddef>`).
`difference_type`	Type that can represent the difference of two iterators for the container. Usually just `ptrdiff_t` (from `<cstddef>`).
`allocator_type`	The allocator type with which the template was instantiated.

The container adapters define only `value_type` and `size_type` from the preceding table. They add `container_type`, which is the type of the underlying container. The `bitset` defines none of these types. The `string` defines all of these types, and adds the `traits_type`.

Common Iterator Methods

All sequential and associative containers in the STL, plus the `string`, define the following methods for obtaining iterators into the container. The container adapters and the `bitset` do not support iteration over their elements.

Prototype	Description	Running Time

`iterator begin();`
 	
`const_iterator begin() const;`
	Returns an `iterator` (or `const_iterator`) referring to the first element in the container.	Constant

`iterator end();`
 	
`const_iterator end() const;`
	Returns an `iterator` (or `const_iterator`) referring to the “past-the-end” element in the container.	Constant

`reverse_iterator rbegin();`
 	
`const_reverse_iterator rbegin() const;`
	Returns an `iterator` (or `const_iterator`) referring to the last element in the container.	Constant

`reverse_iterator rend();`
 	
`const_reverse_iterator rend() const;`
	Returns an `iterator` (or `const_iterator`) referring to the “past-the-beginning” element in the container.	Constant

Note that the `const` versions of the methods will execute on `const` objects, while the non-`const` versions will execute on non-`const` objects.

Common Comparison Operators

All containers in the STL except for the `priority_queue` support the standard comparisons, usually implemented as global overloaded operators. If `C` and `D` are two objects of the same container (with the same template parameters), then the following comparisons are valid:

Comparison	Description	Running Time
`C == D`	`C` and `D` have the same number of elements, and all the elements are equal.	Linear
`C != D`	`! (C == D)`	Linear
`C < D`	Calls `lexicographical_compare()` on the ranges of elements in the two containers.	Linear
`C > D`	`D < C`	Linear
`C <= D`	`! (C > D)`	Linear
`C >= D`	`! (C < D)`	Linear

Note that the `bitset` provides different comparison operations, described below.

Other Common Functions and Methods

The following table shows other functions and methods that are common to all the sequential and associative containers. Note that not all the shared methods are shown here; some are discussed for each individual container below.

Prototype	Description	Running Time
`allocator_type get_allocator() const;`	Returns the allocator used by the container	Constant

The `string` also provides a `get_allocator()` method.

Note on Running Time

Unless otherwise stated, all running time statements are with regard to the number of elements in the container on which the method is called. Some operations are relative to other factors such as the number of elements inserted into or erased from a container. In those cases, the running time statement is explicitly qualified with an extra explanation. In order to avoid writing "the number of elements in the container on which the method is called," S denotes that phrase in the tables that follow.

Sequential Containers

The sequential containers include `vector`, `deque`, and `list`.

vector

This section describes all the `public` methods on the `vector`, as defined in the `<vector>` header file.

Iterator

The `vector` provides random-access iteration.

Template Definition

```
template <typename T, typename Allocator = allocator<T> >
```

`T` is the element type to be stored in the vector, and `Allocator` is the type of allocator to be used by the vector.

Constructors, Destructors, and Assignment Methods

Prototype	Description	Running Time	Invalidates Iterators and References?
`explicit vector(const Allocator& =Allocator());`	Default constructor; constructs a `vector` of size `0`, optionally with the specified allocator	Constant	N/A
`explicit vector(size_type n, const T& value = T(), const Allocator& =Allocator());`	Constructs a `vector` of size `n`, with optional allocator, and initializes elements to `value`	Linear in the number of elements inserted (`n`)	N/A
`template <typename InputIterator> vector(InputIterator first, InputIterator last, const Allocator& =Allocator());`	Constructs a `vector`, with optional allocator, and inserts the elements from `first` to `last`. It's a method template in order to work on any iterator	Linear in the number of elements inserted	N/A
`vector(const vector<T, Allocator>& x)`	Copy constructor	Linear in the size of `x`	N/A
`~vector()`	Destructor	Linear (destructor is called on every element in the `vector`)	Yes
`vector<T, Allocator>& operator=(const vector<T, Allocator>& x);`	Assignment operator	Linear in S plus the size of `x`	Yes
`template <class InputIterator> void assign(InputIterator first, InputIterator last);`	Removes all the current elements and inserts all the elements from `first` to `last`	Linear in S plus number of elements inserted	Yes
`void assign(size_type n, const T& u);`	Removes all the current elements and inserts `n` elements of value `u`	Linear in S plus `n`	Yes
`void swap(vector<T, Allocator>&);`	Swaps the contents of the two `vector`s	Usually constant (just swaps internal pointers),but not required by the standard	No (but the iterators and references now refer to elements in a different container)

Adding and Deleting Elements

vectors provide several ways to insert and delete elements. insert() and push_back() allocate memory as needed to store the new elements.

Prototype	Description	Running Time	Invalidates Iterators and References?
void push_back(const T& x);	Inserts element x at the end of the vector.	Amortized constant	Yes (if reallocation occurs)
void pop_back();	Removes the last element in the vector.	Constant	Only iterators and references referring to that element
iterator insert(iterator position, const T& x);	Inserts the element x before the element at position, shifting all subsequent elements to make room. Returns an iterator referring to the element inserted.	Amortized constant at the end; linear elsewhere	Yes (all if reallocation occurs; otherwise only those referring to elements at or past the insertion point)
void insert(iterator position, size_type n, const T& x);	Inserts n copies of x before the element at position, shifting all subsequent elements to make room.	Amortized constant at the end; linear in S plus n elsewhere	Yes (all if reallocation occurs; otherwise only those referring to elements at or past the insertion point)
template <typename InputIterator> void insert(iterator position, InputIterator first, InputIterator last);	Inserts all elements from first to last before the element at position.	Amortized constant at the end; linear in S plus number of elements inserted elsewhere	Yes (all if reallocation occurs; otherwise only those referring to elements at or past the insertion point)
iterator erase(iterator position);	Removes the element at position, shifting subsequent elements to remove the gap. Returns an iterator referring to the element following the one that was erased.	Constant at the end; linear elsewhere	Invalidate all iterators and references at or past position

Prototype	Description	Running Time	Invalidates Iterators and References?
`iterator erase(iterator first, iterator last);`	Removes the elements from `first` to `last`. Returns an iterator referring to the element following the ones that were erased.	Constant at the end; linear elsewhere	Invalidates all iterators and references at or past first
`void clear();`	Erases all elements in the `vector`.	Linear	Yes

Accessing Elements

`vector`s provide standard array access syntax as well as methods for retrieving values at specific locations. All these methods provide both `const` and non-`const` versions. If called on a `const` vector, the `const` version of the method is called, which returns a `const_reference` to the element at that location. Otherwise, the non-`const` version is called, which returns a `reference` to the element at that location.

Prototype	Description	Running Time	Invalidates Iterators and References?

`reference operator[](size_type n);`
 	
`const_reference operator[](size_type n) const;`
	Array syntax for element access. Does not perform bounds checking.	Constant	No

`reference at(size_type n);`
 	
`const_reference at(size_type n) const;`
	Method for element access. Throws `out_of_range` if `n` refers to a nonexistent element.	Constant	No

`reference front();`
 	
`const_reference front() const;`
	Returns a reference to the first element. Undefined if there are no elements.	Constant	No

`reference back();`
 	
`const_reference back() const;`
	Returns a reference to the last element. Undefined if there are no elements.	Constant	No

Retrieving and Setting Size and Capacity

Prototype	Description	Running Time	Invalidates Iterators and References?
`size_type size() const;`	The number of elements in the `vector`.	Usually constant, but not required to be by the standard	No
`size_type max_size() const;`	The maximum number of elements the `vector` could hold. Not usually a very useful method, because the number is likely to be quite large.	Usually constant, but not required to be by the standard	No
`void resize(size_type sz, T c = T());`	Changes the number of elements in the `vector` (the size) to `sz`, creating new ones with the default constructor if required. Can cause a reallocation and can change the capacity.	Linear	Yes (if reallocation occurs)
`size_type capacity() const;`	The number of elements the `vector` could hold without a reallocation.	Usually constant (but unspecified)	No
`bool empty() const;`	Returns `true` if the `vector` currently has no elements; `false` otherwise.	Constant	No
`void reserve(size_type n);`	Changes the capacity of the `vector` to `n`. Does not change the size of the `vector`.	Linear	Yes (if reallocation occurs)

The vector<bool> Specialization

The partial specialization of `vector<bool>` provides all the methods found in `vector`, with a few differences from a normal instantiation.

reference Class

Instead of a `typedef` for `reference`, `vector<bool>` provides a `reference` class that serves as a proxy for the `bool`. All methods that return `references` (such as `operator[]` and `at[]`) return a proxy object.

Bit Methods

The `vector<bool>` provides one new method on both the container and the reference type:

Prototype	Description	Running Time	Invalidates Iterators and References?
`flip()`	If called on the container, negates all the elements. If called on a reference object, negates that element.	Linear on container; constant on reference object.	No

deque

This section describes all the `public` methods on the `deque`, as defined in the `<deque>` header file.

Iterator

The `deque` provides random-access iteration.

Template Definition

```
template <typename T, typename Allocator = allocator<T> >
```

`T` is the element type to be stored in the `deque`, and `Allocator` is the type of allocator to be used by the `deque`.

Constructors, Destructors, and Assignment Methods

Prototype	Description	Running Time	Invalidates Iterators and References?
`explicit deque(const Allocator& =Allocator());`	Default constructor; constructs a `deque` of size `0`, optionally with the specified allocator.	Constant	N/A
`explicit deque(size_type n, const T& value = T(), const Allocator& =Allocator());`	Constructs a `deque` of size `n` and initializes elements to value.	Linear in `n`	N/A
`template <typename InputIterator> deque(InputIterator first, InputIterator last, const Allocator& =Allocator());`	Constructs a `deque` and inserts the elements from `first` to `last`. It's a method template in order to work on any iterator.	Linear in number of elements inserted	N/A

Table continued on following page

Prototype	Description	Running Time	Invalidates Iterators and References?
`deque(const deque<T, Allocator>& x)`	Copy constructor.	Linear in size of x	N/A
`~deque()`	Destructor.	Linear (destructor is called on every element in the deque)	Yes
`deque<T, Allocator>& operator=(const deque<T, Allocator>& x);`	Assignment operator.	Linear in S plus size of x	Yes
`template <class InputIterator> void assign(InputIterator first, InputIterator last);`	Removes all the current elements and inserts all the elements from `first` to `last`.	Linear in S plus number of elements inserted	Yes
`void assign(size_type n, const T& u);`	Removes all the current elements and inserts `n` elements of value `u`.	Linear in S plus `n`	Yes
`void swap(deque<T, Allocator>&);`	Swaps the contents of the two `deque`s.	Usually constant (just swaps internal pointers), but not required by the standard	No (but the iterators and references now refer to elements in a different container)

Adding and Deleting Elements

`deque`s provide several ways to insert and delete elements. `insert()`, `push_back()`, and `push_front()` allocate memory as needed to store the new elements.

Prototype	Description	Running Time	Invalidates Iterators and References?
`void push_back(const T& x);`	Inserts element `x` at the end of the `deque`.	Constant	Iterators: yes References: no
`void push_front(const T& x);`	Inserts element `x` at the beginning of the `deque`.	Constant	Iterators: yes References: no

Prototype	Description	Running Time	Invalidates Iterators and References?
`void pop_back();`	Removes the last element in the `deque`.	Constant	Invalidates only iterators and References to the last element
`void pop_front();`	Removes the first element in the `deque`.	Constant	Invalidates only iterators and references to the first element
`iterator insert(iterator position, const T& x);`	Inserts the element `x` before the element at `position`. Returns an iterator referring to the element inserted.	Linear in the middle; constant at beginning or end	Yes, unless the element is added to the front or back; then, only iterators are invalidated
`void insert(iterator position, size_type n, const T& x);`	Inserts `n` copies of `x` before the element at `position`.	Linear in S plus n when inserting in the the middle; constant at beginning or end	Yes, unless the elements are added to the front or back; then, only iterators are invalidated
`template <typename InputIterator> void insert(iterator position, InputIterator first, InputIterator last);`	Inserts all elements from `first` to `last` before the element at `position`.	Linear in S plus number of elements inserted when inserting in the middle; constant at beginning or end	Yes, unless the elements are added to the front or back; then, only iterators are invalidated
`iterator erase(iterator position);`	Removes the element at `position`. Returns an iterator referring to the element following the one that was erased.	Linear in the middle; constant at beginning or end	Yes, unless the erased element is at the front or back; then, only iterators and references to that element are invalidated
`iterator erase(iterator first, iterator last);`	Removes the elements from `first` to `last`. Returns an iterator referring to the element following the ones that were erased.	Linear in the middle; constant at beginning or end	Yes, unless the erased elements are at the front or back; then only iterators and references to that element are invalidated
`void clear();`	Erases all elements in the `deque`.	Linear	Yes

Accessing Elements

deques provide standard array access syntax as well as methods for retrieving values at specific locations. All these methods provide both const and non-const versions. If called on a const deque, the const version of the method is called, which returns a const_reference to the element at that location. Otherwise, the non-const version is called, which returns a reference to the element at that location.

Prototype	Description	Running Time	Invalidates Iterators and References?
reference operator[](size_type n); const_reference operator[](size_type n) const;	Array syntax for element access. Does not perform bounds checking.	Constant	No

reference at(size_type n);
 	
const_reference at(size_type n) const;
	Method for element access. Throws out_of_range if n refers to a nonexistent element.	Constant	No

reference front();
 	
const_reference front() const;
	Returns a reference to the first element. Undefined if there are no elements.	Constant	No

reference back();
 	
const_reference back() const;
	Returns a reference to the last element. Undefined if there are no elements.	Constant	No

Retrieving and Setting Size

Prototype	Description	Running Time	Invalidates Iterators and References?
size_type size() const;	The number of elements in the deque.	Usually constant, but not required to be by the standard	No
size_type max_size() const;	The maximum number of elements the deque could hold. Not usually a very useful method, as the number is likely to be quite large.	Usually constant, but not required to be by the standard	No

Prototype	Description	Running Time	Invalidates Iterators and References?
`void resize(size_type sz, T c = T());`	Changes the number of elements in the `deque` (the size) to `sz`, creating new ones with the default constructor if required.	Linear	Yes
`bool empty() const;`	Returns `true` if the `deque` currently has no elements; `false` otherwise.	Constant	No

list

This section describes all the `public` methods on the `list`, as defined in the `<list>` header file.

Iterator

The list provides bidirectional iteration.

Template Definition

```
template <typename T, typename Allocator = allocator<T> >
```

`T` is the element type to be stored in the `list`, and `Allocator` is the type of allocator to be used by the `list`.

Constructors, Destructors, and Assignment Methods

Prototype	Description	Running Time	Invalidates Iterators and References?
`explicit list(const Allocator& =Allocator());`	Default constructor; constructs a list of size `0`, optionally with the specified allocator.	Constant	N/A
`explicit list(size_type n, const T& value = T(), const Allocator& =Allocator());`	Constructs a list of size `n` and initializes elements to `value`.	Linear in `n`	N/A

Table continued on following page

Prototype	Description	Running Time	Invalidates Iterators and References?
`template <typename InputIterator> list(InputIterator first, InputIterator last, const Allocator& =Allocator());`	Constructs a `list` and inserts the elements from `first` to `last`. It's a method template in order to work on any iterator.	Linear in number of elements inserted	N/A
`list(const list<T, Allocator>& x)`	Copy constructor.	Linear in size of `x`	N/A
`~list()`	Destructor.	Linear (destructor is called on every element in the list)	Yes
`list<T, Allocator>& operator=(const list<T, Allocator>& x);`	Assignment operator.	Linear in S plus size of `x`	Yes
`template <class InputIterator> void assign(InputIterator first, InputIterator last);`	Removes all the current elements and inserts all the elements from `first` to `last`.	Linear in S plus number of elements inserted	Yes
`void assign(size_type n, const T& u);`	Removes all the current elements and inserts `n` elements of value `u`.	Linear in S plus `n`	Yes
`void swap(list<T, Allocator>&);`	Swaps the contents of the two `list`s.	Usually constant (just swaps internal pointers), but not required by the standard	No (but the iterators and references now refer to elements in a different container)

Adding and Deleting Elements

The list provides constant-time operations for adding and deleting elements.

Prototype	Description	Running Time	Invalidates Iterators and References?
`void push_back(const T& x);`	Inserts element `x` at the end of the `list`.	Constant	No
`void push_front(const T& x);`	Inserts element `x` at the beginning of the `list`.	Constant	No

Prototype	Description	Running Time	Invalidates Iterators and References?
`void pop_back();`	Removes the last element in the `list`.	Constant	Only those referring to the erased element
`void pop_front();`	Removes the first element in the `list`.	Constant	Only those referring to the erased element
`iterator insert(iterator position, const T& x);`	Inserts the element `x` before the element at `position`. Returns an `iterator` referring to the element inserted.	Constant	No
`void insert(iterator position, size_type n, const T& x);`	Inserts `n` copies of `x` before the element at `position`.	Constant in S; Linear in n	No
`template <typename InputIterator> void insert(iterator position, InputIterator first, InputIterator last);`	Inserts all elements from `first` to `last` before the element at `position`.	Constant in S; linear in number of elements inserted	No
`iterator erase(iterator position);`	Removes the element at `position`. Returns an `iterator` referring to the element following the one that was erased.	Constant	Only those referring to the erased element
`iterator erase(iterator first, iterator last);`	Removes the elements from `first` to `last`. Returns an `iterator` referring to the element following the ones that were erased.	Constant in S; linear in number of elements erased	Only those referring to the erased elements
`void clear();`	Erases all elements in the `list`.	Linear	Yes

Accessing Elements

The `list` does not provide random access to elements.

Prototype	Description	Running Time	Invalidates Iterators and References?

`reference front();`
 	
`const_reference front() const;`
	Returns a `reference` to the first element. Undefined if there are no elements.	Constant	No

`reference back();`
 	
`const_reference back() const;`
	Returns a `reference` to the last element. Undefined if there are no elements.	Constant	No

Retrieving and Setting Size

Prototype	Description	Running Time	Invalidates Iterators and References?
`size_type size() const;`	The number of elements in the `list`.	Usually constant, but not required to be by the standard	No
`size_type max_size() const;`	The maximum number of elements the `list` could hold. Not usually a very useful method, as the number is likely to be quite large.	Usually Constant, but not required to be by the standard	No
`void resize(size_type sz, T c = T());`	Changes the number of elements in the `list` (the size) to sz, creating new ones with the default constructor if required.	Linear	Yes
`bool empty() const;`	Returns `true` if the `list` currently has no elements; `false` otherwise.	Constant	No

List Operations

The `list` container provides several specialized operations that are either not covered in the generalized algorithms or are more efficient than the equivalent algorithm.

Prototype	Description	Running Time	Invalidates Iterators and References?
`void splice(iterator position, list<T, Allocator>& x);`	Inserts the `list` `x` into position. Destroys `x`.	Constant	Invalidates iterators and references to `x`. Does not invalidate those to the `list` on which the method is called
`void splice(iterator position, list<T,Allocator>& x,iterator i);`	Inserts element `i` from list `x` into `position`. Removes element `i` from `x`.	Constant	Invalidate iterators and references only to the element referred to by `i`
`void splice(iterator position, list<T,Allocator>& x, iterator first, iterator last);`	Assumes that `first` and `last` are iterators into `x`. Inserts the specified range from `x` into `position`. Removes the range from `x`.	Constant	Invalidates iterators and references to the elements that are moved

`void remove(const T& value);`
 	
`template <class Predicate> void remove_if(Predicate pred);`
	Removes all elements from the `list` equal to `value` or for which `pred` is `true`.	Linear	Invalidates iterators and references to the erased elements

`void unique();`
 	
`template <class BinaryPredicate> void unique (BinaryPredicate binary_pred);`
	Removes duplicate consecutive elements from the `list`. Duplicates are checked with `operator==` or `binary_pred`.	Linear	Invalidates iterators and references to the erased elements

`void merge(list<T, Allocator>& x);`
 	
`template <class Compare> void merge (list<T,Allocator>& x, Compare comp);`
	Merges `x` into the `list` on which the method is called. Both `list`s must be sorted to start. `x` is empty after the merge. Compares elements with `operator<` or `comp`.	Linear	Invalidate all iterators and references to elements in `x`

`void sort();`
 	
`template <class Compare> void sort(Compare comp);`
	Performs stable sort on elements in the `list`, using `operator<` or the specified `comp` to order elements.	Linear logarithmic	No

`void reverse();`	Reverses the order of the elements in the `list`.	Linear	No

Container Adapters

The container adapters include `stack`, `queue`, and `priority_queue`. Container adapters do not support iterators.

stack

This section describes all the public methods on the `stack`, as defined in the `<stack>` header file.

Template Definition

```
template <typename T, typename Container = deque<T> >
```

`T` is the element type to be stored in the `stack`, and `Container` is the sequential container on which it is based. The `Container` can be `vector`, `deque`, or `list`.

Constructors, Destructors, and Assignment Methods

The `stack` provides only default and single-argument constructors. It provides no destructor, copy constructor, or assignment operator because its only data member, the underlying container, handles all of that.

Prototype	Description	Running Time
`explicit stack(const Container& =Container());`	Default or 1-argument constructor; constructs a `stack` using the specified container to store its elements	Constant

Adding and Deleting Elements

`stack`s provide one way to add elements and one way to remove them.

Prototype	Description	Running Time
`void push(const T& x);`	Inserts element `x` at the top of the `stack`	Constant
`void pop();`	Removes the top element from the `stack`	Constant

Accessing Elements

Prototype	Description	Running Time
`T& top();` `const T& top() const;`	Retrieves (but does not remove) the top value in the `stack`. Returns a `const` reference if called on a `const` object; otherwise returns a reference.	Constant

Retrieving Size

Prototype	Description	Running Time
`size_type size() const;`	The number of elements the `stack`	Usually constant, but in not required to be by the standard
`bool empty() const;`	Returns `true` if the `stack` currently has no elements; `false` otherwise	Constant

queue

This section describes all the public methods on the `queue`, as defined in the `<queue>` header file.

Template Definition

```
template <typename T, typename Container = deque<T> >
```

`T` is the element type to be stored in the `queue`, and `Container` is the sequential container on which it is based. The `Container` can be `deque` or `list`. `vector` does not qualify because it does not provide constant-time insertion and removal at both ends of the sequence.

Constructors, Destructors, and Assignment Methods

The `queue` provides only default and single-argument constructors. It provides no destructor, copy constructor, or assignment operator because its only data member, the underlying container, handles all of that.

Prototype	Description	Running Time
`explicit queue(const Container& =Container());`	Default or 1-argument constructor; constructs a `queue` using the specified container to store its elements	Constant

Adding and Deleting Elements

The `queue` provides one way to add elements and one way to remove them.

Prototype	Description	Running Time
`void push(const T& x);`	Inserts element `x` at the end of the `queue`	Constant
`void pop();`	Removes the front element from the `queue`	Constant

Accessing Elements

Prototype	Description	Running Time
`T& front();` `const T& front() const;`	Retrieves (but does not remove) the front element in the `queue`. Returns a `const` reference if called on a `const` object; otherwise returns a reference.	Constant
`T& back();` `const T& back() const;`	Retrieves (but does not remove) the back element in the `queue`. Returns a `const` reference if called on a `const` object; otherwise returns a reference.	Constant

Retrieving Size

Prototype	Description	Running Time
`size_type size() const;`	The number of elements in the `queue`.	Usually constant, but not required to be by the standard
`bool empty() const;`	Returns `true` if the `queue` currently has no elements; `false` otherwise.	Constant

priority_queue

This section describes all the `public` methods on the `priority_queue`, as defined in the `<queue>` header file.

Template Definition

```
template <typename T, typename Container = vector<T>, typename Compare =
    less<typename Container::value_type> >;
```

`T` is the element type to be stored in the `queue`, `Container` is the sequential container on which it is based, and `Compare` is the type of the comparison object or function to be used for comparing elements in the `priority_queue`. The `Container` can be `vector` or `deque`. `list` does not qualify because it does not provide random access to its elements.

Constructors, Destructors, and Assignment Methods

The `priority_queue` provides no destructor, copy constructor, or assignment operator because its underlying container handles all of that.

Prototype	Description	Running Time
explicit priority_queue(const Compare& x = Compare(), const Container& =Container());	Constructs a queue using the specified comparison callback, and the specified container to store its elements.	Linear in the number of elements in the specified container.
template <class InputIterator> priority_queue (InputIterator first, InputIterator last, const Compare& x = Compare(), const Container& = Container());	Constructs a queue using the specified comparison function, and the specified container to store its elements. Inserts elements from first to last into the container.	Linear in the number of elements in the container and the range first to last.

Adding and Deleting Elements

The priority_queue provides one way to add elements and one way to remove them.

Prototype	Description	Running Time
void push(const T& x);	Inserts element x in its priority order in the queue	Logarithmic
void pop();	Removes the element with highest priority from the queue	Logarithmic

Accessing Elements

Prototype	Description	Running Time
const T& top() const;	Retrieves (but does not remove) a const reference to the front element in the priority_queue	Constant

Retrieving Size

Prototype	Description	Running Time
`size_type size() const;`	The number of elements in the `priority_queue`	Usually constant, but not required to be by the standard
`bool empty() const;`	Returns `true` if the `priority_queue` currently has no elements; `false` otherwise	Constant

Associative Containers

The associative containers include `map`, `multimap`, `set`, and `multiset`.

Associative Container Typedefs

Associative containers add the following `typedefs` to the common container `typedefs`.

Type Name	Description
`key_type`	The key type with which the container is instantiated.
`key_compare`	The comparison class or function pointer type with which the container is instantiated.
`value_compare`	Class for comparing two `value_type` elements. For `sets` this is the same is `key_compare`. For `maps`, it must compare the key/value pairs.
`mapped_type`	(`maps` and `multimaps` only) The "value" type with which the container is instantiated.

map and multimap

This section describes all the `public` methods on the `map` and the `multimap`, as defined in the `<map>` header file. The `map` and `multimap` provide almost identical operations. The main differences are that the `multimap` allows duplicate elements with the same key and doesn't provide `operator[]`.

The template definition, constructors, destructor, and assignment operator show the **`map`** *versions. The* **`multimap`** *versions are identical, but with the name* **`multimap`** *instead of* **`map`**. *The text of the method descriptions uses "***`map`***" to mean both* **`map`** *and* **`multimap`**. *When there is a distinction,* **`multimap`** *is used explicitly.*

Iterators

The `map` and `multimap` provide bidirectional iteration.

Template Definition

```
template <typename Key, typename T, typename Compare = less<Key>,
    typename Allocator = allocator<pair<const Key, T> > > class map;
```

`Key` is the key type and `T` is the value type for elements to be stored in the `map`. `Compare` is the comparison class or function pointer type for comparing keys. `Allocator` is the type of allocator to be used by the `map`.

Constructors, Destructors, and Assignment Methods

Prototype	Description	Running Time	Invalidates Iterators and References?
`explicit map(const Compare& comp = Compare(), const Allocator& =Allocator());`	Default constructor; constructs a `map` of size `0`, optionally with the specified comparison object and allocator.	Constant	N/A
`template <typename InputIterator> map(InputIterator first, InputIterator last, const Compare& comp = Compare(), const Allocator& =Allocator());`	Constructs a `map` and inserts the elements from `first` to `last`. It's a method template in order to work on any iterator.	Linear logarithmic in the number of elements inserted; linear if the inserted element range is already sorted according to `comp`	N/A
`map(const map<key, T, Compare, Allocator>& x)`	Copy constructor.	Linear in the size of `x`	N/A
`~map()`	Destructor.	Linear (destructor is called on every element in the vector)	Yes
`map<key, T, Compare, Allocator>& operator=(const map<Key, T, Compare, Allocator>& x);`	Assignment operator.	Linear in S plus the size of `x`	Yes
`void swap(map<Key, T, Compare, Allocator>& );`	Swaps the contents of the two `map`s.	Usually constant (just swaps internal pointers), but not required by the standard	No (but the iterators and references now refer to elements in a different container)

Adding and Deleting Elements

Inserting an element into the container consists of adding a key/value `pair`. It allocates memory for the `pair`.

Prototype	Description	Running Time	Invalidates Iterators and References?
`pair<iterator, bool> insert(const value_type& x);`	Not supported for `multimap`. Inserts the key/value `pair` `x` if and only if the `map` does not already contain an element with that key. Returns a `pair` of an `iterator` referring to the element with the key of `x` and a `bool` specifying whether the insertion actually took place.	Logarithmic	No
`iterator insert(const value_type& x);`	`multimap` only. Inserts the key/value `pair` `x`. Returns an `iterator` referring to the element with the key of `x`.	Logarithmic	No
`iterator insert(iterator position, const value_type&  x);`	Inserts the key/value `pair` `x`. For `multimap`s, always inserts it. For `map`s, inserts if and only if the `map` does not already contain an element with that key. Returns an `iterator` referring to the element with the key of `x`. `position` is only a hint to the `map`.	Usually logarithmic, but amortized; constant if position is correct	No
`template <typename InputIterator> void insert(InputIterator first, InputIterator last);`	Inserts elements from `first` to `last`. For `multimap`s inserts all elements. For `map`s inserts only those for which there is not already a key/value `pair` with that key.	Usually N log ($S + N$), where N is the number of elements inserted, but linear if the inserted range is already sorted correctly	No

Prototype	Description	Running Time	Invalidates Iterators and References?
`void erase(iterator position);`	Removes the element at position.	Amortized constant	Invalidate only iterators and references to the erased element
`size_type erase(const key_type& x);`	Removes all elements in the container with key `x` and returns the number of elements removed.	Logarithmic	Invalidate only iterators and references to the erased element
`void erase(iterator first, iterator last);`	Removes the elements from `first` to `last`.	log(S + *N*), where *N* is the number of elements erased	Invalidate only iterators and references to the erased elements
`void clear();`	Erases all elements in the `map`.	Linear	Yes

Accessing Elements

Most of the methods in this category have `const` and non-`const` versions. If called on a `const map`, the `const` version of the method is called, which returns a `const_reference` or `const_iterator`. Otherwise, the non-`const` version is called, which returns a `reference` or `iterator`.

Prototype	Description	Running Time	Invalidates Iterators and References?
`T& operator[](const key_type& x);`	`map`s, but not `multimap`s, provide standard array access syntax. If no element exists with the specified key, a new element is inserted with that key.	Logarithmic	No

`iterator find(const key_type& x);`
 	
`const_iterator find(const key_type& x) const;`
	Returns an `iterator` referring to an element with key matching `x`.
	If no element has key `x`, returns `end()`. Note that for `multimap`s the returned `iterator` can refer to any one of the elements with the specified key.
		Logarithmic	No

`size_type count(const key_type& x) const;`	Returns the number of elements with key matching `x`.	Logarithmic	No

Table continued on following page

Prototype	Description	Running Time	Invalidates Iterators and References?

`iterator lower_bound(const key_type& x);`
 	
`const_iterator lower_bound(const key_type& x) const;`
	Returns an `iterator` referring to the first element whose key is greater than or equal to `x`. Can return `end()` if all elements have keys less than `x`.	Logarithmic	No

`iterator upper_bound(const key_type& x);`
 	
`const_iterator upper_bound(const key_type& x) const;`
	Returns an `iterator` referring to the first element whose key is greater than `x`. Can return `end()` if all elements have keys less than or equal to `x`.	Logarithmic	No

`pair<iterator, iterator> equal_range(const key_type& x);`
 	
`pair<const_iterator, const_iterator> equal_range(const key_type& x) const;`
	Combination of `lower_bound()` and `upper_bound()`. Returns a `pair` of `iterators` referring to the first and one-past-the-last elements with keys matching `x`. If the two `iterators` are equal, there are no elements with key `x`.	Logarithmic	No

Retrieving Size and Comparison Objects

Prototype	Description	Running Time	Invalidates Iterators and References?
`size_type size() const;`	The number of elements in the `map`.	Usually constant, but not required to be by the standard	No
`size_type max_size() const;`	The maximum number of elements the `map` could hold. Not usually a very useful method, as the number is likely to be quite large.	Usually constant, but not required to be by the standard	No

Prototype	Description	Running Time	Invalidates Iterators and References?
`bool empty() const;`	Returns `true` if the `map` currently has no elements; `false` otherwise	Constant	No
`key_compare key_comp() const;`	Returns the object used to compare the keys	Constant	No
`value_compare value_comp() const;`	Returns the object used to compare elements in the `map` by comparing the keys in the key/ value `pair` object	Constant	No

set and multiset

This section describes all the `public` methods on the `set` and the `multiset`, as defined in the `<set>` header file. The `set` and `multiset` provide almost identical operations. The main difference is that the `multiset` allows duplicate elements with the same key.

*The template definition, constructors, destructor, and assignment operator show the **`set`** versions. The **`multiset`** versions are identical, but with the name **`multiset`** instead of **`set`**. The text of the method descriptions uses "**`set`**" to mean both **`set`** and **`multiset`**. When there is a distinction, **`multiset`** is used explicitly.*

Iterators

The `set` and `multiset` provide bidirectional iteration.

Template Definition

```
template <typename Key, typename Compare = less<Key>,
    typename Allocator = allocator<Key> > class set;
```

`Key` is the type of the elements to be stored in the `set`. `Compare` is the comparison class or function pointer type for comparing keys. `Allocator` is the type of allocator to be used by the `set`.

Constructors, Destructors, and Assignment Methods

Prototype	Description	Running Time	Invalidates Iterators and References?
`explicit set(const Compare& comp = Compare(), const Allocator& =Allocator());`	Default constructor; constructs a `set` of size `0`, optionally with the specified comparison object and allocator.	Constant	N/A
`template <typename InputIterator> set(InputIterator first, InputIterator last, const Compare& comp = Compare(), const Allocator& =Allocator());`	Constructs a `set` and inserts the elements from `first` to `last`. It's a method template in order to work on any iterator.	Linear logarithmic in the number of elements inserted; linear if the inserted element range is already sorted according to comp	N/A
`set(const set<Key, Compare, Allocator>& x)`	Copy constructor.	Linear in the size of `x`	N/A
`~set()`	Destructor.	Linear (destructor is called on every element in the `set`)	Yes
`set<Key, Compare, Allocator>& operator=(const map<Key, Compare, Allocator>& x);`	Assignment operator.	Linear in S plus the size of `x`	Yes
`void swap(set<Key, Compare, Allocator>&);`	Swaps the contents of the two `set`s.	Usually constant (just swaps internal pointers), but not required by the standard	No (but the iterators and references now refer to elements in a different container)

Adding and Deleting Elements

Inserting an element into the `set` allocates memory for the element.

Prototype	Description	Running Time	Invalidates Iterators and References?
`pair<iterator, bool> insert(const value_type& x);`	Not supported for `multiset`. Inserts the element `x` if and only if the `set` does not already contain that element. Returns a `pair` of an `iterator` referring to the element and a `bool` specifying whether the insert actually took place.	Logarithmic	No
`iterator insert(const value_type& x);`	`multiset` only. Inserts the element `x`. Returns an `iterator` referring to the element `x`.	Logarithmic	No
`iterator insert(iterator position, const value_type&  x);`	Inserts the element `x`. For `multisets`, always inserts it. For `sets`, inserts if and only if the `set` does not already contain that element. Returns an `iterator` referring to the element. `position` is only a hint to the `set`.	Usually logarithmic, but amortized constant if position is correct	No
`template <typename InputIterator> void insert (InputIterator first, InputIterator last);`	Inserts elements from `first` to `last`. For `multisets` inserts all elements. For `set` inserts only those for which there is not already an element equal to the element to be inserted.	Usually N log (S + N), where `N` is the number of elements inserted, but linear if the inserted range is already sorted correctly	No
`void erase(iterator position);`	Removes the element at `position`.	Amortized constant	Invalidate only iterators and references to the erased element
`size_type erase(const key_type& x);`	Removes all elements in the container matching `x` and returns the number of elements removed.	Logarithmic	Invalidate only iterators and references to the erased element

Table continued on following page

Prototype	Description	Running Time	Invalidates Iterators and References?
`void erase(iterator first, iterator last);`	Removes the elements from `first` to `last`.	log(S + *N*), where N is the number of elements erased	Invalidate only iterators and references to the erased elements
`void clear();`	Erases all elements in the `set`.	Linear	Yes

Accessing Elements

The methods in this category have only `const` versions that return `iterator` (not `const_iterator`).

Prototype	Description	Running Time	Invalidates Iterators and References?
`iterator find(const key_type& x) const;`	Returns an `iterator` referring to an element matching `x`. If no element matches `x`, returns `end()`. Note that for `multisets` the returned `iterator` can refer to any one of the elements matching `x`.	Logarithmic	No
`size_type count(const key_type& x) const;`	Returns the number of elements matching `x`.	Logarithmic	No
`iterator lower_bound(const key_type& x) const;`	Returns an `iterator` referring to the first element greater than or equal to `x`. Can return `end()` if all elements are less than `x`.	Logarithmic	No
`iterator upper_bound(const key_type& x) const;`	Returns an `iterator` referring to the first element greater than `x`. Can return `end()` if all elements are less than or equal to `x`.	Logarithmic	No

Prototype	Description	Running Time	Invalidates Iterators and References?
`pair<iterator, iterator> equal_range(const key_type& x) const;`	Combination of `lower_bound()` and `upper_bound()`. Returns a `pair` of `iterator`s referring to the first and one-past the last elements matching `x`. If the two `iterator`s are equal, there are no elements matching `x`.	Logarithmic	No

Retrieving Size and Comparison Objects

Prototype	Description	Running Time	Invalidates Iterators and References?
`size_type size() const;`	The number of elements in the `set`.	Usually constant, but not required to be by the standard	No
`size_type max_size() const;`	The maximum number of elements the `set` could hold. Not usually a very useful method, as the number is likely to be quite large.	Usually constant, but not required to be by the standard	No
`bool empty() const;`	Returns `true` if the `set` currently has no elements; `false` otherwise.	Constant	No
`key_compare key_comp() const;` `value_compare value_comp() const;`	Returns the object used to compare the elements.	Constant	No

bitset

This section describes all the public methods on the bitset, as defined in the <bitset> header file. The bitset provides no support for iteration, and the standard does not provide any running time guarantees.

Template Definition

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

N is the number of bits in the bitset.

Constructors

Several method prototypes have been simplified slightly to make them less confusing.

Prototype	Description	Exceptions
`bitset();`	Default constructor; constructs a bitset of size N (as specified in the template parameter) and initializes all bits to zero.	None
`bitset(unsigned long val);`	Constructs a bitset of size N and initializes the bits to the bits in val. If N is larger than the number of bits in val, the extra high-order bits are initialized to 0. If N is smaller than the number of bits in val, the extra high-order bits of val are ignored.	None
`explicit bitset (const string& str, string:: size_type pos = 0, string::size_type len = max_len);`	Constructs a bitset of size N and initializes it from the string str, starting at pos, for len characters. The characters in str must all be either 0 or 1. If len < N, extra high-order bits initialized to 0.	out_of_range if pos > str.size() invalid_argument if any character in str is not 0 or 1

Bit Manipulation Methods

These methods modify the bitset on which they are called. All the bit manipulation methods return a reference to the bitset object.

<table>
<tr><th>Prototype</th><th>Description</th><th>Exceptions</th></tr>
<tr><td>bitset<N>& set();

bitset<N>&
set(size_t pos,
int val = true);</td><td>Sets all the bits in the set to true, or sets the specified bit to the specified value</td><td>out_of_range if pos is not a valid position in the bitset</td></tr>
<tr><td>bitset<N>& reset();

bitset<N>&
reset(size_t pos);</td><td>Sets all the bits, or the specified bit, to false</td><td>out_of_range if pos is not a valid position in the bitset</td></tr>
<tr><td>bitset<N>& flip();

bitset<N>&
flip(size_t pos);</td><td>Toggles all the bits or the specified bit</td><td>out_of_range if pos is not a valid position in the bitset</td></tr>
<tr><td>reference
operator[]
(size_t pos);</td><td>Returns a read/write reference to the bit at pos that can be used to set the bit to true or false, set the bit to the value of another bit, call flip() on the bit, or negate the bit with ~</td><td>out_of_range if pos is not a valid position in the bitset</td></tr>
</table>

Overloaded Bitwise Operators

The normal bitwise operators are overloaded for the bitset. Note that the global functions are actually templates to handle bitsets of any size N, but the prototypes in this table have been simplified slightly for clarity.

<table>
<tr><th>Prototype</th><th>Description</th><th>Exceptions</th></tr>
<tr><td>bitset<N>
operator&(const
bitset<N>&, const
bitset<N>&);

bitset<N> operator|
(const bitset<N>&,
const bitset<N>&);

bitset<N> operator^
(const bitset<N>&,
const bitset<N>&</td><td>Global functions providing bitwise and, or, and exclusive or. The operands are not modified, and the functions return a new bitset containing the result.</td><td>None</td></tr>
<tr><td>bitset<N> operator<<
(size_t pos) const;

bitset<N> operator>>
(size_t pos) const;</td><td>Methods providing bitwise left-shift and right-shift operations. Fills in exposed bits with false. Doesn't modify bitset on which it's called.</td><td>None</td></tr>
<tr><td>bitset<N> operator~()
const;</td><td>Method providing bitwise not. Doesn't modify bitset on which it's called.</td><td>None</td></tr>
</table>

Table continued on following page

Prototype	Description	Exceptions

`bitset<N>& operator&=(const bitset<N>& rhs);`
 	
`bitset<N>& operator|=(const bitset<N>& rhs);`
 	
`bitset<N>& operator^=(const bitset<N>& rhs);`
 	
`bitset<N>& operator<<=(size_t pos);`
 	
`bitset<N>& operator>>=(size_t pos);`
	Methods providing bitwise assignment operators. Modify the `bitset` on which they are called.	None

Stream Operators

The `bitset` provides stream-related functions. The prototypes have been simplified slightly for this table (they are actually function templates).

Prototype	Description	Exceptions

`ostream& operator<<(ostream& os, const bitset<N>& x);`
 	
`istream& operator>>istream& os, bitset<N>& x);`
	Global insertion and extraction operators for `bitset`s. Reads and writes `bitset`s as strings of `0` or `1`.	None, unless the streams throw

Obtaining Information about Bits in the Bitset

Prototype	Description	Exceptions

`bool operator[] (size_t pos) const;`
 	
`bool test(size_t pos) const;`
	Returns the value of the bit at `pos`
	
	`bitset`
		`out_of_range` if `pos` is not a valid position in the

`size_t size() const;`	Returns the number of bits in the `bitset`	None
`size_t count() const;`	Returns the number of bits in the `bitset` set to true	None

Prototype	Description	Exceptions
`bool any() const;`	Returns `true` if one or more bits in the `bitset` are set to true	None
`bool none() const;`	Returns `true` if all bits in the `bitset` are set to false	None

`bool operator==(const bitset<N>& rhs) const;`
 	
`bool operator!=(const bitset<N>& rhs) const;`
	Bit-by-bit comparison between two `bitsets`	None

`unsigned long to_ulong() const;`
 	
`string to_string() const;`
	Converts the `bitset` to an `unsigned long` or to a `string`	None

string

The `string` and `wstring` classes are `typedef`s for `char` and `wchar_t` instantiations of the `basic_string` class template. For simplicity, this section shows the methods on the `string`, which uses the `char` type. Keep in mind that they apply to any instantiation of the `basic_string` template. The `basic_string`, `string`, and `wstring` are defined in the `<string>` header file.

The standard specifies no running time guarantees for the string methods.

Iterator

The `string` provides random-access iteration.

Note on Exceptions

Most string operations throw `length_error` if the resultant `string` size would exceed its maximum size. You don't usually need to worry about this case, so we omit it from the exception lists in the following tables.

Note on npos

`npos` is a constant that stands for "no position." When used as a default value for parameters specifying size it implies unlimited size (up to the length of the `string`). When used as a default parameter for positions, it implies the end of the `string`. When returned from `find()` and similar methods, it means that no match was found.

Constructors, Destructors, and Assignment Methods

Prototype	Description	Invalidates Iterators and References?	Exceptions
`explicit string(const Allocator& a =Allocator());`	Default constructor; constructs a `string` of size `0`, optionally with the specified allocator.	N/A	None (unless the allocator throws an exception)

`string(const char* s, size_type n, const Allocator& a = Allocator());`
 	
`string(const char* s, const Alocator& a = Allocator());`
	Constructs a `string` containing the c-style string `s`. The first form inserts `n` characters from s, regardless of the `\0` character. The second form inserts characters from `s` until the `\0` character.	N/A	None (unless the allocator throws an exception)

`string(size_type n, char c, const Allocator&a = Allocator());`	Constructs a string containing `n` copies of character `c`.	N/A	None (unless the allocator throws an exception)
`template <typename InputIterator> string (InputIterator first, InputIterator last, const Allocator& =Allocator());`	Constructs a `string` and inserts the elements from `first` to `last`. It's a method template in order to work on any iterator.	N/A	None (unless the allocator throws an exception)

`string(const string& str);`
 	
`string(const string& str, size_type pos, size_type n = npos, const Allocator& a = Allocator());`
	Copy constructor. The second form initializes the new string to contain the first `n` characters of `str`, starting at `pos`.	N/A	`out_of_range` if `pos > str.size()`

`~string()`	Destructor.	Yes	None

Prototype	Description	Invalidates Iterators and References?	Exceptions

string& operator=(const string& str);
 	
string& operator=(const char* s);
 	
string& operator=(char c);
	Assignment operator: replaces the contents of the target string object with a copy of str, s, or c.	Yes	None

template <class InputIterator> string& assign (InputIterator first, InputIterator last);	Remove all the current elements and inserts all the elements from first to last.	Yes	None

string& assign (const string& str);
 	
string& assign(const string& str, size_type pos, size_type n);
 	
string& assign(const char* s, size_type n);
 	
string& assign(const char* s);
 	
string& assign(size_type n, char c);
	Similar to the assignment operator: replaces the contents of the target string object with a copy of str or s, or n copies of c.	Yes	out_of_range if pos > str.size()

void swap(string& str);	Swaps the contents of the two strings.	Yes	None

Adding and Deleting Characters

Prototype	Description	Invalidates Iterators and References?	Exceptions
iterator insert(iterator p, char c);	Inserts the character c before the character at p, shifting all subsequent characters to make room. Returns an iterator referring to the character inserted.	Yes	None
void insert(iterator p, size_type n, char c);	Inserts n copies of c before the character at p, shifting all subsequent characters to make room.	Yes	None
template <typename InputIterator> void insert(iterator p, InputIterator first, InputIterator last);	Inserts all characters in the range [first,last) before the character at p.	Yes	None

string& insert(size_type pos, const string& str);
 	
string& insert(size_type pos, const char* s);
	Inserts all the characters from str or s starting at position pos.	Yes	out_of_range if pos > size()

string& insert(size_type pos, const string& str, size_type pos2, size_type n);
 	
string& insert(size_type pos, const char* s, size_type n);
	Inserts n characters from str or s starting at position pos in the target string. The first form requires a starting position in the source string also.	Yes	out_of_range if pos > size() or pos2 > str.size()

string& insert(size_type pos, size_type n, char c);	Inserts n copies of c starting at pos in the target string.	Yes	None
string& erase(size_type pos = 0, size_type n = npos);	Removes n characters starting at pos.	Yes	out_of_range if pos > size()

Prototype	Description	Invalidates Iterators and References?	Exceptions
`iterator erase(iterator position);`	Removes the character at `position`. Returns an `iterator` referring to the element following the one that was erased.	Yes	None
`iterator erase(iterator first, iterator last);`	Removes the characters in the range [`first`,`last`). Returns an `iterator` referring to the element following the ones that were erased.	Yes	None
`void clear();`	Removes all characters in the `string`.	Yes	None

`string& replace(size_type pos, size_type n1, const string& str);`
`string& replace(size_type pos, size_type n1, const char* s);`
`string& replace(iterator i1, iterator i2, const string& str);`
`string& replace(iterator i1, iterator i2, const char* s);`
	The first two forms remove `n1` characters from the target `string` starting as position `pos`. The last two forms remove the characters in the range [`i1`, `i2`). All forms insert the contents of `str` or `s` at `pos` or `i1`.	Yes	`out_of_range` if `pos > size()` `string& replace(size_type pos, size_type n1, const string& str, size_type pos2, size_type n2);`

`string& replace(size_type pos, size_type n1, const char* s, size_type n2);` `string& replace(iterator i1, iterator i2, const char* s, size_type n2);`	The first two forms remove `n1` characters from the target `string` starting at `pos`. The third form removes the characters in the range [`i1`,`i2`). Inserts `n2` characters from `str` beginning at `pos2`, or from `s`.	Yes	`out_of_range` if `pos > size()` or `pos2 > str.size()`

Table continued on following page

Prototype	Description	Invalidates Iterators and References?	Exceptions

`string& replace(size_type pos, size_type n1, size_type n2, char c);`
 	
`string& replace(iterator i1, iterator i2, size_type n, char c);`
	Removes `n1` characters from the target `string` starting at `pos`, or the characters from the range [`i1`,`i2`). Inserts `n2` copies of `c` at `pos` or `i1`.	Yes	`out_of_range` if `pos > size()`

`template <typename InputIterator> string& replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2);`	Removes the characters in the range [`i1`,`i2`). Inserts the characters in the range [`j1`,`j2`) starting at `i1`.	Yes	None

`string& operator+=(const string& str);`
 	
`string& append(const string& str);`
 	
`string& append(const string& str, size_type pos, size_type n);`
 	
`string& operator+=(const char* s);`
 	
`string& append(const char* s);`
 	
`string& append(const char* s, size_type n);`
 	
`string& operator+=(char c);`
 	
`string& append(size_type n, char c);`
	Both `operator+=()` and `append()` add characters to the end of the target `string` and return a reference to it. The characters to be added can be from a `string`, C-style string, or a character. `append()` allows the client to specify the start position and number of characters from a source `string`, number of characters from a source C-style string, or number of copies of a single character to append.	Yes	`out_of_range` if `pos > str.size()`

Prototype	Description	Invalidates Iterators and References?	Exceptions
`template <typename InputIterator> string& append(InputIterator first, InputIterator last);`	Adds the characters in the range [`first`,`last`) to the end of the target string.	Yes	None
`void push_back(char c);`	Appends character `c` to the target `string`.	Yes	None

Accessing Characters

The access methods provide both `const` and non-`const` versions. If called on a `const string`, the `const` version of the method is called, which returns a `const_reference` to the character at that location. Otherwise, the non-`const` version is called, which returns a `reference` to the character at that location.

Prototype	Description	Invalidates Iterators and References?	Exceptions
`reference operator[] (size_type n);` `const_reference operator[] (size_type n) const;`	Array syntax for character access. Does not perform bounds checking.	No	None
`reference at(size_type n);` `const_reference at(size_type n) const;`	Method for character access.	No	`out_of_range` if `n >= size()`

Obtaining Characters

Prototype	Description	Invalidates Iterators and References?	Exceptions
`const char* c_str() const;`	Returns a C-style string containing all the characters in the target `string`. The returned string is valid only until a non-`const` method is called on the target `string`.	No	None

Table continued on following page

Prototype	Description	Invalidates Iterators and References?	Exceptions
`const char* data() const;`	Identical to `c_str()`, except that the return value points to a character array without a terminating `\0`.	No	None
`size_type copy(char* s, size_type n, size_type pos = 0) const;`	Copies into the character buffer pointed to by `s` `n` characters in the target `string` starting at `pos`.	No	`out_of_range` if `pos > size()`
`string substr(size_type pos = 0, size_type n = npos) const;`	Returns a new string containing *n* characters from the target string starting at pos.	No	`out_of_range` if `pos > size()`

Retrieving and Setting Size and Capacity

None of the following methods throw an exception.

Prototype	Description	Invalidates Iterators and References?
`size_type size() const;` `size_type length() const;`	The number of characters in the `string`.	No
`size_type max_size() const;`	The maximum number of characters the `string` could hold. Not usually a very useful method, as the number is likely to be quite large.	No
`void resize(size_type sz, char c = char());`	Changes the number of characters in the `string` (the size) to `sz`, creating new ones with the default constructor if required. Can cause a reallocation and can change the capacity.	Yes
`size_type capacity() const;`	The number of characters the `string` could hold without a reallocation.	No
`bool empty() const;`	Returns `true` if the `string` currently has no characters; `false` otherwise.	No
`void reserve(size_type n);`	Changes the capacity of the `string` to `n`. Does not change the size of the `string`.	Yes

Searching

Prototype	Description	Invalidates Iterators and References?	Exceptions

`size_type find(char c, size_type pos = 0) const;`
 	
`size_type rfind(char c, size_type pos = npos) const;`
	Returns the index of the first character in the target `string` matching `c`. The search starts at `pos` and moves forward with `find()` or backward with `rfind()`. Returns `npos` if no match is found.	No	None

`size_type find(const string& str, size_type pos = 0) const;`
 	
`size_type find(const char* s,size_type pos, size_type n) const;`
 	
`size_type find(const char* s, size_type pos = 0);`
 	
`size_type rfind(const string& str, size_type pos = npos) const;`
 	
`size_type rfind(const char* s, size_type pos, size_type n) const;`
 	
`size_type rfind(const char* s, size_type pos = npos);`
	Returns the index of the beginning character of the first substring in the target string matching `str` or `s`. The search starts at `pos` and moves forward with `find()` or backward with `rfind()`. Two forms allow you to specify that only `n` characters of the C-style string `s` should be matched. Returns `npos` if no match is found.	No	None

`size_type find_first_of(const string& str, size_type pos = 0) const;`
 	
`size_type find_first_of(const char* s, size_type pos = 0) const;`
 	
`size_type find_first_of(const char* s, size_type pos, size_type n) const;`
 	
`size_type find_last_of(const string& str, size_type pos = npos) const;`
	Returns the index of the first character that is in `str` or `s`. The search starts at `pos` and moves forward with `find_first_of()` or backward with `find_last_of()`. Two forms allow you to specify that only `n` characters of the C-style string `s` should be matched. Returns `npos` if no match is found.	No	None

Table continued on following page

Prototype	Description	Invalidates Iterators and References?	Exceptions
`size_type find_last_of(const char* s, size_type pos = npos) const;size_type find_last_of(const char* s, size_type pos, size_type n) const;`			

`size_type find_first_not_of(const string& str, size_type pos = 0) const;`
 	
`size_type find_first_not_of(const char* s, size_type pos = 0) const;`
 	
`size_type find_first_not_of(const char* s, size_type pos, size_type n) const;`
 	
`size_type find_last_not_of(const string& str, size_type pos = npos) const;`
 	
`size_type find_last_not_of(const char* s, size_type pos = npos) const;`
 	
`size_type find_last_not_of(const char* s, size_type pos, size_type n) const;`
	Like `find_first_of()` and `find_last_of()`, except finds the first character not in `str` or `s`.	No	None

Comparisons

In addition to the standard comparison operators such as `operator==` and `operator!=`, strings provide a `compare()` method with the following prototypes:

Prototype	Description	Invalidates Iterators and References?	Exceptions
`int compare(const string& str) const;` `int compare(const char* s) const;`	Performs a character-by-character comparison between `str` or `s` and the target `string`. Returns `0` if the two `string`s are equal, `< 0` if the target string is lexicographically less than `str` or `s`, or `> 0` if `str` or `s` is lexicographically less than the target `string`.	No	None
`int compare(size_type pos1, size_type n1, const string& str) const;` `int compare(size_type pos1, size_type n1, const char* s) const;`	Like the previous forms of `compare()`, except allows the caller to specify a start position in the target string and number of characters to compare.	No	`out_of_range` if `pos1 > size()`
`int compare(size_type pos1, size_type n1, const string& str, size_type pos2, size_type n2) const;` `int compare(size_type pos1, size_type n1, const char* s, size_type n2) const;`	Like the previous forms of `compare()`, except allows the caller to specify a start position in `str` and a number of characters in `str` or `s`.	No	`out_of_range` if `pos1 > size()` or `pos2 > str.size()`

Concatenating strings

Prototype	Description	Invalidates Iterators and References?	Exceptions

`string operator+(const string& lhs, const string& rhs);`
 `string operator+(const char* lhs, const string& rhs);`
 `string operator+(char lhs, const string& rhs);`
 `string operator+(const string& lhs, char* rhs);`
 `string operator+(const string& lhs, char rhs);`
	Concatenates two strings and returns the result. One of the source strings must be a `string` object, but the other can be a `string` object, C-style string, or single character. Note that these are global functions, not methods.	No	None

Streaming strings

Prototype	Description	Invalidates Iterators and References?	Exceptions
`istream& operator>> (istream& is, string& str);`	Reads characters from `is` and appends them to `str` until end-of-file or a space character is found.	Yes	None
`ostream& operator<< (ostream& os, const string& str);`	Writes the characters in `str` to `os`.	No	None

`istream& getline(istream& is, string& str);`
 	
`istream& getline(istream& is, string& str, char delim);`
	Reads characters from `is` and appends them to `str` until '\n' or `delim` is found. `delim` is not appended to `str`.	Yes	None

Algorithms

All the algorithms are templatized functions on one or more type parameters. For simplicity, the tables in this section don't show the template part of the function prototype. Instead, they use the following type names to refer to templatized types:

Type Name	Meaning
`T`	Element type.
`InputIterator, InputIterator1, InputIterator2`	An iterator that is "at least" input.
`ForwardIterator, ForwardIterator1, ForwardIterator2`	An iterator that is "at least" forward.
`OutputIterator, OutputIterator1, OutputIterator2`	An iterator that is "at least" output.
`BidirectionalIterator, BidirectionalIterator1, BidirectionalIterator2`	An iterator that is "at least" bidirectional.
`RandomAccessIterator, RandomAccessIterator1, RandomAccessIterator2`	A random access iterator.

Type Name	Meaning
`Compare`	A function pointer or functor that compares two elements, returning `true` if the first is less than the second, `false` otherwise.
`Predicate`	A function pointer or functor that returns `true` or `false` when passed an element as its single argument.
`BinaryPredicate`	A function pointer or functor that returns `true` or `false` when passed two elements. Usually used to compare two elements such that it returns `true` when they are equal, `false` otherwise.
`UnaryOperation`	A function pointer or functor that takes an element and returns an element.
`BinaryOperation`	A function pointer or functor that takes two elements and returns a single element.
`Function`	A function pointer or functor that takes one element. The return type is irrelevant.
`RandomNumberGenerator`	A function pointer or functor that takes one integer argument `n` and returns an integer in the range `[0,n)`.
`Generator`	A function pointer or functor that takes no arguments and returns an element.
`Size`	An integral type.

All functions are declared in `<algorithm>` unless otherwise noted.

Utility Algorithms

Algorithm Prototype	Algorithm Synopsis	Return Value	Running Time

`const T& min(const T& a, const T& b);`
 	
`const T& min(const T& a, const T& b, Compare comp);`
 	
`const T& max(const T& a, const T& b);`
 	
`const T& max(const T& a, const T& b, Compare comp);`
	Return the minimum or maximum of two values, using `operator<` or the supplied comparison callback to compare them	Returns a reference to the minimum or maximum value	Constant

`void swap(T& a, T& b);`	Exchanges two values	`void`	Constant

Nonmodifying Algorithms

The nonmodifying algorithms do not change the elements in the range or ranges on which they operate.

Search Algorithms

Algorithm Prototype	Algorithm Synopsis	Return Value	Requires Sorted Sequence?	Running Time

`InputIterator find (InputIterator first, InputIterator last, const T& value);`
 	
`InputIterator find_if (InputIterator first, InputIterator last, Predicate pred);`
	Finds the first element that matches `value` with `operator==` or for which `pred` returns `true`.	An iterator referring to the first matching element, or `last` if no match is found.	No	Linear

`ForwardIterator1 find_first_of (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);`
 	
`ForwardIterator1 find_first_of (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);`
	Searches in the range [`first1`,`last1`) for any one of the elements in the range [`first2`,`last2`). The elements are compared with `operator==` or `pred`.	An iterator referring to the first matching element, or `last1` if no match is found.	No	Quadratic

`ForwardIterator adjacent_find (ForwardIterator first, ForwardIterator last);`
 	
`ForwardIterator adjacent_find (ForwardIterator first, ForwardIterator last, BinaryPredicate pred);`
	Finds the first instance of two consecutive elements in the range [`first`,`last`) that are equal to each other, compared with `operator==` or `pred`.	An iterator referring to the first element of the matching pair, or last if no match is found.	No	Linear

Algorithm Prototype	Algorithm Synopsis	Return Value	Requires Sorted Sequence?	Running Time
`ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2), ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred), ForwardIterator1 find_end (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2), ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred),`	Finds the first (`search()`) or last (`find_end()`) subsequence in the range [`first1`,`last1`) that matches the subsequence [`first2`,`last2`).	An iterator referring to the first element of the matching subsequence in the range [`first1`,`last1`), or `last1` if no matching subsequence is found.	No	Quadratic
`ForwardIterator search_n (ForwardIterator first, ForwardIterator last, Size count, const T& value);` `ForwardIterator search_n(ForwardIterator first, ForwardIterator last, Size count, const T& value, BinaryPredicate pred);`	Finds the first subsequence of *n* consecutive elements equal to `value` in the range [`first`,`last`). Compares elements with `operator==` or `pred`.	An iterator referring to the first element of the matching subsequence, or `last` if no matching subsequence is found.	No	Quadratic

Table continued on following page

Algorithm Prototype	Algorithm Synopsis	Return Value	Requires Sorted Sequence?	Running Time

`ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const T& value);`
 	
`ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);`
 	
`ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const T& value);`
 	
`ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);`
 	
`pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first, ForwardIterator last, const T& value);`
 	
`pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first, ForwardIterator`
	Finds the beginning (`lower_bound()`) end (`upper_bound()`) or both sides (`equal_range()`) of the range including `value`. Compares elements with `operator` < or `comp`.	`lower_bound()` returns an iterator referring to the first element greater than or equal to value, or last if all elements are less than `value`. `upper_bound()` returns an iterator referring to the first element greater than `value`, or `last` if all elements are less than `value`. `equal_range()` returns the `last, const T& value, Compare comp);` `pair` of iterators that `lower_bound()` and `upper_bound()` would return separately.	Yes	Logarithmic for random access
				
				containers; linear otherwise.

`bool binary_search (ForwardIterator first, ForwardIterator last, const T& value);`
 	
`bool binary_search (ForwardIterator first, ForwardIterator last, const T& value, Compare comp);`
	Finds `value` in a sorted range `[first,last)`.	Returns `true` or `false` specifying whether `value` is in the range `[first,last)`.	Yes	Logarithmic for random access iterators; Linear otherwise.

Algorithm Prototype	Algorithm Synopsis	Return Value	Requires Sorted Sequence?	Running Time

`ForwardIterator min_element(ForwardIterator first, ForwardIterator last);`
 	
`ForwardIterator min_element(ForwardIterator first, ForwardIterator last, Compare comp);`
 	
`ForwardIterator max_element(ForwardIterator first, ForwardIterator last);`
 	
`ForwardIterator max_element(ForwardIterator first, ForwardIterator last, Compare comp);`
	Finds the minimum or maximum element in the range [`first`,`last`), comparing elements with `operator` < or `comp`.	Returns an iterator referring to the minimum or maximum element.	No	Linear

Numerical Processing Algorithms

The algorithms `accumulate()`, `inner_product()`, `partial_sum()`, and `adjacent_difference()` are in `<numeric>`.

Algorithm Prototype	Algorithm Synopsis	Return Value	Running Time

`int count(InputIterator first, InputIterator last, const T& value);`
 	
`int count_if(InputIterator first, InputIterator last, Predicate pred);`
	Counts the number of elements matching `value` with `operator==` or for which `pred` returns `true`.	The number of matching elements	Linear

`T accumulate(InputIterator first, InputIterator last, T init);`
 	
`T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation binary_op);`
	"Accumulates" the values of all the elements in a sequence starting with `init`. Accumulates elements with `operator+` or `binary_op`.	The accumulated value	Linear

Table continued on following page

Algorithm Prototype	Algorithm Synopsis	Return Value	Running Time

`T inner_product (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init);`
 	
`T inner_product (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init, BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);`
	Similar to `accumulate()`, but works on two sequences. Calls `operator*` or `binary_op2` on parallel elements in the two sequence, accumulating the result with `operator+` or `binary_op2`. If the two sequences represent mathematical vectors, the algorithm calculates the dot product of the vectors. The range starting at `first2` should be at least as long as the range [`first1`,`last1`).	The accumulated value	Linear

`OutputIterator partial_sum (InputIterator first, InputIterator last, OutputIterator result);`
 	
`OutputIterator partial_sum (InputIterator first, InputIterator last, OutputIterator result, BinaryOperation binary_op);`
	Writes to each element of `result` the parallel element in the range [`first`,`last`), plus the sum of all preceding elements in the range [`first`,`last`). `result` can be the same as `first` (in which case it's not technically a nonmodifying algorithm). The sum can be calculated with `operator+` or `binary_op`.	The past-the-end iterator of the sequence starting at `result`	Linear

`OutputIterator adjacent_difference (InputIterator first, InputIterator last, OutputIterator result);`
 	
`OutputIterator adjacent_difference (InputIterator first, InputIterator last, OutputIterator result, BinaryOperation binary_op);`
	Writes to each element of `result` (except the first) the parallel element in the range [`first`,`last`) minus the preceding element in the range [`first`,`last`). The first element in `result` is assigned the value referred to by `first`. `result` can be the same as `first` (in which case it's not technically a nonmodifying algorithm). The difference can be calculated with `operator-` or `binary_op`.	The past-the-end iterator of the sequence starting at `result`	Linear

Comparison Algorithms

Algorithm Prototpe	Algorithm Synopsis	Return Value	Running Time

`bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);`
 	
`bool equal (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate pred);`
	Determines if two sequences are equal by checking if they have the same elements in the same order. Elements are compared with `operator==` or `pred`. The range starting at `first2` must be at least as long as the range [`first1`,`last1`).	`true` if the two ranges are equal; `false` otherwise	Linear

`pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);`
 	
`pair<InputIterator1, InputIterator2> mismatch (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate pred);`
	Returns the first element in each sequence that does not match the element in the same location in the other sequence. Elements are compared with `operator==` or `pred`. The range starting at `first2` must be at least as long as the range [`first1`,`last1`).	A pair of iterators referring into each sequence at the point of mismatch. If no mismatch is found, returns `last1` and the equivalent iterator into the range starting at `first2`	Linear

`bool lexicographical_ compare(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);`
 	
`bool lexicographical_ compare(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp);`
	Compares two sequences to determine their "lexicographical" ordering. Compares each element ofthe first sequence with its equivalent element in the second. If one element is less than the other, that sequence is lexicographically first. If the elements are equal, compares the next elements in order.
	The two ranges need not be the same length. The shorter range is lexicographically less than the longer, if all elements up to that point are equal. Elements are compared with `operator<`.
		`true` if the sequences are lexicographically equal; `false` otherwise	Linear

Operational Algorithms

Algorithm Prototype	Algorithm Synopsis	Return Value	Running Time
`Function for_each Function f) (InputIterator first, InputIterator last,`	Executes `f` on each element in the sequence	`f`, which can be used to accumulate information about the elements	Linear

Modifying Algorithms

Unlike the nonmodifying algorithms, the modifying algorithms modify the elements in the range on which they're called. However, there is usually a form that writes the modified elements to a destination range instead of modifying the source range directly.

Algorithm Prototype	Algorithm Synopsis	Return Value	Running Time

`OutputIterator transform (InputIterator first, InputIterator last, OutputIterator result, UnaryOperation op);`
 	
`OutputIterator transform (InputIterator first, InputIterator last, OutputIterator result, BinaryOperation binary_op);`
	Calls `op` or `binary_op` on each element or pair of elements in the range [`first,last`), storing the results in the range starting at `result`. `result` can be equal to `first` for "in-place" operation. The range starting with `result` must be at least as big as the range [`first,last`).	An iterator referring to one past the end of the new sequence starting with `result`.	Linear

`OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result);`	Copies elements from the range [`first,last`) to the range beginning with `result`. `result` may not be in the range [`first,last`), but the ranges may otherwise overlap.	Returns the past-the-end iterator of the new sequence beginning at `result`.	Linear
`BidirectionalIterator2 copy_backward (BidirectionalIterator1 first, BidirectionalIterator1 last, BidirectionalIterator2 result);`	Copies elements from the range [`first, last`) to the range for which `result` is the past-the-end iterator. `result` may not be in the range [`first,last`), but the ranges may otherwise overlap.	Returns the start iterator of the new sequence that ends with `result`.	Linear

Algorithm Prototype	Algorithm Synopsis	Return Value	Running Time

`void iter_swap (ForwardIterator1 a, ForwardIterator2 b);`
 	
`ForwardIterator2 swap_ranges (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2);`
	Swap two elements referred to by iterators `a` and `b`, or two ranges [`first1`,`last1`) and the range beginning at `first2`.	`swap_ranges()` returns the past-the-end iterator of the range beginning at `first2`.	Linear

`void replace (ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value);`
 	
`void replace_if (ForwardIterator first, ForwardIterator last, Predicate pred, const T& new_value);`
 	
`OutputIterator teplace_copy(InputIterator first, InputIterator last, OutputIterator result, const T& old_value, const T& new_value);`
 	
`OutputIterator replace_copy_if(Iterator first, Iterator last, OutputIterator result, Predicate pred, const T& new_value);`
	Replaces in the range [`first`,`last`) with `new_value` all elements matching `old_value` with `operator==` or for which `pred` returns `true`.
	The first two forms replace in-place.
	The second two forms modify the range beginning with `result`.
	The ranges cannot overlap. The in-place forms return nothing.
		The copy forms return the past-the-end iterator of the new range beginning with `result`.	Linear

`void fill(ForwardIterator first, ForwardIterator last, const T& value);`
 	
`void fill_n(OutputIterator first, Size n, const T& value);`
	Sets all elements in the range [`first`,`last`) or the range [`first`,`first+n`) to `value`.	`void`	Linear

`void generate (ForwardIterator first, ForwardIterator last, Generator gen);`
 	
`void generate_n (OutputIterator first, Size n, Generator gen);`
	Like `fill()` and `fill_n()`, except calls `gen` to generate values.	`void`	Linear

Table continued on following page

Algorithm Prototype	Algorithm Synopsis	Return Value	Running Time

`ForwardIterator remove (ForwardIterator first, ForwardIterator last, const T& value);`
 	
`ForwardIterator remove_if(ForwardIterator first, ForwardIterator last, Predicate pred);`
 	
`OutputIterator remove_copy (InputIterator first, InputIterator last, OutputIterator result, const T& value);`
 	
`OutputIterator remove_copy_if(InputIterator first, InputIterator last, OutputIterator result, Predicate pred);`
	The first two forms "remove" from the range [`first`,`last`) elements that match `value` or for which `pred` returns `true`. Removed elements are copied to the end of the range, and the new end of the (shorter) range is returned. The second two forms are like `copy()`, except they also remove while copying elements matching `value` or for which `pred` returns `true`. The rules that apply to `copy()` apply here.	The past-the-end iterator of the destination range.	Linear

`ForwardIterator unique (ForwardIterator first, ForwardIterator last);`
 	
`ForwardIterator unique (ForwardIterator first, ForwardIterator last, BinaryPredicate pred);`
 	
`OutputIterator unique_copy(InputIterator first, InputIterator last, OutputIterator result);`
 	
`OutputIterator unique_copy(InputIterator first, InputIterator last, OutputIterator result, BinaryPredicate pred);`
	Removes duplicates from the range [`first`,`last`), either in-place or copying results to the range beginning with result. Elements are compared with `operator==` or `pred`.	The past-the-end iterator of the destination range.	Linear

`void reverse (BidirectionalIterator first, BidirectionalIterator last);`
 	
`OutputIterator reverse_copy (BidirecitonalIterator first, BidirectionalIterator last, OutputIterator result);`
	Reverses the order of the elements in the range [`first`,`last`), either in-place or copying the results to the range beginning with `result`. In the second form, the source and destination range should not overlap.	The second form returns the new past-the-end iterator of the range beginning with `result`.	Linear

Algorithm Prototype	Algorithm Synopsis	Return Value	Running Time

`void rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);`
 	
 `OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result);`
	Rotates the elements such that the range [`first`,`middle`) follows the range [`middle`,`last`). `middle` need not be the "true" middle of the range. The second form copies the rotated range to the range starting at `result`. The source and destination ranges should not overlap.	The second form returns the new past-the-end iterator of the range beginning with `result`.	Linear

`bool next_permutation (BidirectionalIterator first, BidirectionalIterator last);`
 	
 `bool next_permutation (BidirectionalIterator first, BidirectionalIterator last, Compare comp);`
 	
 `bool next_permutation (BidirectionalIterator first, BidirectionalIterator last);`
 	
 `bool next_permutation (BidirectionalIterator first, BidirectionalIterator last, Compare comp);`
	Modifies the range [`first`,`last`) by transforming it into its "next" or "previous" permutation. A permutation of elements is "less" than another according to the algorithm in `lexicographical_compare()`, using `operator<` or `comp`. Successive calls to one or the other will permute the sequence into all possible permutations of elements.	Returns `true` if there is another "next" or "previous" permutation.	Linear

Sorting Algorithms

All the sorting algorithms except `partition()`, `stable_partition()`, and `random_shuffle()` have two forms: the first uses `operator <` and the second takes a comparison callback. For brevity, only the first forms are shown in this table. Remember that you can always pass a custom sorting criterion as a final parameter to the function.

Algorithm Prototype	Algorithm Synopsis	Return Value	Running Time

`void sort RandomAccessIterator first, RandomAccessIterator last);`
 	
`void stable_sort RandomAccessIterator first, RandomAccessIterator last);`
	Sorts the range [`first,last`) in-place. `stable_sort()` preserves the order of duplicate elements.	`void`	*N* log *N* in general, but quadratic in worst cast.

`void partial_sort (RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last);`
 	
`RandomAccessIterator partial_sort_copy (InputIterator first, InputIterator last, RandomAccessIterator result_first, RandomAccessIterator result_last);`
	After the call to `partial_sort()`, the range [`first, middle`) will have elements as if the whole range [`first,last`) were sorted. However, the remaining elements in the range [`middle,last`) will not be sorted. `partial_sort_copy()` leaves the range [`first,last`) unchanged, instead copying results to [`result_first, result_last`).	`void` or the past-the-end iterator of the new range.	*N* log *N*.

`void nth_element (RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);`	Relocates the element referred to by `nth` in the range [`first,last`) as if the entire range were sorted. Also partitions the range as if `partition()` had been called.	`void`	Linear in general; quadratic in worst case

`OutputIterator merge(InputIterator1 first1, InputIterator1 last1, InputIterator1 first2, InputIterator2 last2, OutputIterator result);`
 	
`OutputIterator inplace_merge (BidirectionalIterator first, BidirectionalIterator middle, BidriectionalIterator last);`
	The first form merges the two sorted ranges [`first1,last1`) and [`first2,last2`) into the range starting at result. All three ranges must be distinct. The second form merges two consecutive ranges in-place.	The past-the-end iterator of the merged range.	Linear

Algorithm Prototype	Algorithm Synopsis	Return Value	Running Time

`void make_heap (RandomAcessIterator first, RandomAccessIterator last);`
 	
`void push_heap (RandomAccessIterator first, RandomAccessIterator last);`
 	
`void pop_heap (RandomAccessIterator first, RandomAccessIterator last);`
 	
`void sort_heap (RandomAccessIterator first, RandomAccessIterator last);`
	`make_heap()` constructs a heap out of the range [`first`,`last`). `push_heap()` adds the element referred to by last to the heap in the range [`first`,`last-1`). `pop_heap()` removes the element referred to by `first`, and makes a heap out of the remaining elements. After `pop_heap()`, the heap is the range [`first`,`last-1`). Finally, `sort_heap()` turns the heap in [`first`,`last`) into a fully sorted sequence.	`void`	`make_heap()` is linear, `push_heap()` and `pop_heap()` are logarithmic, and `sort_heap()` is *N* log *N*.

`BidirectionalIterator partition (BidirectionalIterator first, BidirectionalIterator last, Predicate pred);`
 	
`BidirectionalIterator stable_partition (BidirectionalIterator first, BidirectionalIterator last, Predicate pred);`
	Sorts the range [`first`,`last`) such that all elements forwhich `pred` returns `true` are before all elements for which it returns `false`. `stable_partition()` preserves the original order of the elements within a partition.	The iterator referring to the element dividing the two partitions. The iterator is the past-the-end iterator of the first partition, and the start iterator of the second.	Linear

`void random_shuffle (RandomAccessIterator first, RandomAccessIterator last);`
 	
`void random_shuffle (RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator& rand);`
	"Unsorts" the elements in the range [`first`,`last`) by randomly reorganizing their order. `random_shuffle()` has an equal chance of generating any of the *n*! orderings of *n* elements. The second form takes a random number generator that must take one integer argument `n` and return an integer in the range [`0`,`n`).	`void`	Linear

Set Algorithms

All the `set` algorithms have two forms: the first uses `operator ==` and the second takes a comparison callback. For brevity, only the first forms are shown in this table. Remember that you can always pass a custom comparison criterion as a final parameter to the function.

Algorithm Prototype	Algorithm Synopsis	Return Value	Running Time

`bool includes`
`(InputIterator1 first1,`
`InputIterator1 last1,`
`InputIterator2 first2,`
`InputIterator2 last2);`
	Determines if the sequence [`first2`,`last2`) is a subset of [`first1`,`last1`).	`true` if the range [`first1`,`last1`) contains all elements in the range [`first2`,`last2`).	Linear

`OutputIterator set_union`
`(InputIterator1 first1,`
`InputIterator1 last1,`
`InputIerator2 first2,`
`InputIterator2 last2,`
`OutputIterator result);`
 	
`OutputIterator`
`set_intersection`
`(InputIterator1 first1,`
`InputIterator1 last1,`
`InputIerator2 first2,`
`InputIterator2 last2,`
`OutputIterator result);`
 	
`OutputIterator`
`set_difference`
`(InputIterator1 first1,`
`InputIterator1 last1,`
`InputIerator2 first2,`
`InputIterator2 last2,`
`OutputIterator result);`
 	
`OutputIterator`
`set_symmetric_difference`
`(InputIterator1 first1,`
`InputIterator1 last1,`
`InputIerator2 first2,`
`InputIterator2 last2,`
`OutputIterator result);`
	Perform the specified set operations on two ranges. The resulting elements are copied to the range starting with result.	The past-the-end iterator of the result range.	Linear

Streams

The hierarchy of `stream` base classes in C++ exhibits the diamond shape common to cases of multiple inheritance. As shown in Figure 1, `istream` and `ostream` are both subclasses of `ios`, and `iostream` is both an `istream` and an `ostream`.

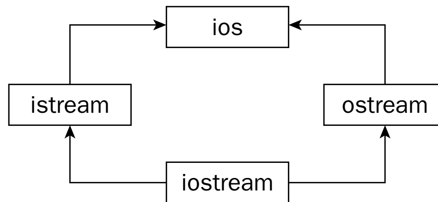

Figure 1

The `istream`, `ostream`, and `iostream` classes serve as base classes to the stream classes used most commonly by C++ programmers. `istringstream` and `ifstream` are both subclasses of `istream`. `ostringstream` and `ofstream` are both subclasses of `ostream`. `stringstream` and `fstream` are both subclasses of `iostream`.

*The information in this section refers to the **`char`**-based stream classes. There are also predefined wide character stream classes, such as **`wios`**, **`wistream`**, and so on. These classes are simply different instantiations of the stream template classes.*

Predefined iostream Objects

Object	Description
`std::cin`	Standard input (usually keyboard). Tied to `std::cout` so that when input is requested with `std::cin`, `std::cout` will be flushed.
`std::cout`	Standard output (usually console).
`std::cerr`	Standard error, unbuffered (flushes immediately).
`std::clog`	Standard error, buffered.
`std::wcin`	Standard input (wide characters).
`std::wcout`	Standard output (wide characters).
`std::wcerr`	Standard error (wide characters).
`std::wclog`	Standard error (wide characters), buffered.

Predefined stream Manipulators

Manipulators are used to change a run-time behavior of a stream and are further described in Chapter 14.

Manipulator	Description
`boolalpha`	Boolean values are displayed/read as "true" and "false."
`noboolalpha`	Boolean values are displayed/read as 1 and 0 (default).
`showbase`	Displays the numeric base of integer output.
`noshowbase`	Will not display the numeric base of integer output (default).
`showpoint`	Floating-point output will always have a decimal point.
`noshowpoint`	Turns off the `showpoint` feature (default).
`showpos`	Prefixes a positive number with a plus sign.
`noshowpos`	Turns off the `showpos` feature (default).
`skipws`	Skips leading white space for input (default for formatted input).
`noskipws`	Turns off the `skipws` feature (default for unformatted input).
`uppercase`	Outputs in uppercase characters.
`nouppercase`	Turns off the `uppercase` feature (default).
`unitbuf`	Instructs the stream to automatically flush after every output operation.
`nounitbuf`	Turns off the `unitbuf` feature (default).
`internal right`	Pads the front (left) of the output with fill characters.
`left`	Pads the back (right) of the output with fill characters.
`dec`	Reads/writes integers in decimal format.
`hex`	Reads/writes integers in hexadecimal format.
`oct`	Reads/writes integers in octal format.
`fixed`	Writes floating point numbers with fixed-point notation (default).
`scientific`	Writes floating-point numbers with scientific notation.

ios

The `ios` class contains data and functionality common to all streams.

Stream Status Methods

Method	Description
`bool good() const`	Returns `true` if the stream is in a usable state (no error bits have been set)
`bool eof() const`	Returns `true` if the stream has reached the end of the file/input (the `eofbit` has been set)
`bool fail() const`	Returns `true` if the stream is in a bad state or the previous operation has failed
`bool bad() const`	Returns `true` if the stream is in a bad state
`operator void*() const`	Equivalent to `good()`
`bool operator!() const`	Equivalent to `fail()`
`void clear()`	Restores the `stream` to working condition by clearing any existing error bits

istream

The input stream base class adds a number of features and new manipulators.

Manipulators

Manipulator	Description
`ws`	Extracts characters until the end of the input or a nonwhite-space character is reached

Formatted Input

Method	Description

`operator>>(short& val)`
`operator>>(unsigned short& val)`
`operator>>(int& val)`
`operator>>(unsigned int& val)`
`operator>>(long& val)`
`operator>>(unsigned long& val)`
`operator>>(float& val)`
`operator>>(double& val)`
`operator>>(long double& val)`
`operator>>(bool& val)`
`operator>>(void*& val)`
	Parses numerical/Boolean data from the `stream`

`operator>>(char* s)`
`operator>>(unsigned char* s)`
`operator>>(signed char* s)`
	Parses character array (C-style `string`) data from the stream

`operator>>(char& c)`
`operator>>(unsigned char& c)`
`operator>>(signed char& c)`
	Parses character data from the stream

Unformatted Input

Method	Description
`int_type get()`	Extracts a single character if one exists. The result will be `eof`.
`istream& get(char& c)`	Extracts a single character and returns a reference to the `stream`.
`istream& get(char* s, streamsize n, char delim)`	Extracts up to `n-1` characters into the character array pointed to by `s` until end of file or the character designated by `delim` is reached. If `delim` is reached, it is not extracted.
`istream& get(char* s, streamsize n)`	Extracts up to `n-1` characters into the character array pointed to by `s` until end of file is reached.
`istream& getline(char* s, streamsize n, char_type delim)` `istream& getline(char* s, streamsize n)`	Similar to `get()` except that the `delim` character is extracted and thrown away. The failure bit is set when the line exceeds `n-1`. In the version with no `delim`, the character `\n` is used.

Method	Description
`istream& read(char* s, streamsize n)`	Extracts *n* characters from the stream into the buffer pointed to by s. `read()` is often used to extract binary data.

`istream& ignore(streamsize n, int_type delim)`
`istream& ignore(streamsize n)`
`istream& ignore()`
	Like `read()`, except it doesn't store the characters read anywhere. Reads `n` characters from stream until `delim` is reached. Removes `delim` from stream. Default `delim` is `eof`, and default number of characters is 1.

Input Stream Navigation

Method	Description
`int_type peek()`	Returns the next character without extracting it
`istream& putback(char c)`	Places the character `c` back on the input stream
`istream& unget()`	Puts the last extracted character back on the input stream
`pos_type tellg()`	Returns the current position of the stream
`istream& seekg(post_type pos)`	Moves to the specified position in the stream
`istream& seekg(off_type& off, pos_type pos)`	Moves to a relative position in the stream from another position

ostream

Many of the output `stream` methods are analogous to input `stream` methods.

Manipulators

Method	Description
`endl`	Outputs an end of line character and then flushes the stream
`ends`	Outputs a null character
`flush`	Calls `flush()`

Formatted Output

Method	Description

`operator<<(short& val)`
`operator<<(unsigned short val)`
`operator<<(int val)`
`operator<<(unsigned int val)`
`operator<<(long val)`
`operator<<(unsigned long val)`
`operator<<(float val)`
`operator<<(double val)`
`operator<<(long double val)`
`operator<<(bool val)`
`operator<<(const void* val)`
	Outputs numerical/Boolean data to the stream

`operator>>(char* s)`
`operator>>(unsigned char* s)`
`operator>>(signed char* s)`
	Outputs character array (C-style string) data to the stream

`operator>>(char& c)`
`operator>>(unsigned char& c)`
`operator>>(signed char& c)`
	Outputs character data to the stream

Unformatted Output

Method	Description
`ostream& put(char c)`	Puts a single character onto the stream
`ostream& write(char* s, streamsize n)`	Puts `n` characters from the buffer pointed to by `s` onto the stream
`ostream& flush()`	Buffered stream data is sent to the output device

Output Stream Navigation

Method	Description
`int_type tellp()`	Returns the current position of the stream
`ostream& seekp(pos_type& pos)`	Moves to the specified position in the stream
`ostream& seekp(off_type& off, ios::seekdir dir)`	Moves `off` characters in the specified direction