1

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 12.

CONTAINERS

The STL divides its containers into four categories. The sequential containers include the `vector`, `list`, `deque`, `array`, and `forward_list`. The container adapters include the `stack`, `queue`, and `priority_queue`. The associative containers include the `map`, `multimap`, `set`, and `multiset`. The unordered associative containers, also called hash tables, include the `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`. Additionally, the `bitset` and `string` can be considered STL containers as well.

Common Typedefs

Most sequential, associative, and unordered 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 `typedef`s 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.

continues

(continued)

TYPE NAME	DESCRIPTION
`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`	A type for iterating over elements of the container.
`const_iterator`	A version of `iterator` for iterating over `const` elements of the container.
`reverse_iterator`	A type 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`, `size_type`, `reference`, and `const_reference` 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, associative, and unordered 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.

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;
```

Returns: An `iterator` (or `const_iterator`) referring to the first element in the container.
Running time: Constant

```
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;
```

Returns: An `iterator` (or `const_iterator`) referring to the "past-the-end" element in the container.
Running time: Constant

```
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
const_reverse_iterator crbegin() const noexcept;
```

Returns: A `reverse_iterator` (or `const_reverse_iterator`) referring to the last element in the container.
Running time: Constant

```
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
const_reverse_iterator crend() const noexcept;
```

Returns: A `reverse_iterator` (or `const_reverse_iterator`) referring to the "past-the-beginning" element in the container.
Running time: 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

Most containers in the STL support the standard comparisons, usually implemented as global overloaded operators. The `priority_queue` does not support them, and the unordered associative containers only support `operator==` and `operator!=`. 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 later.

Other Common Functions and Methods

The following list 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 later.

```
allocator_type get_allocator() const noexcept;
```

Returns: The allocator used by the container.
Running time: 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 this chapter.

Sequential Containers

The sequential containers include `vector`, `deque`, `list`, `array`, and `forward_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 <class T, class Allocator = allocator<T> > class vector;
```

`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

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

Default constructor; constructs a `vector` of size 0, optionally with the specified allocator.
Running time: Constant

```
explicit vector(size_type n);
```

Constructs a `vector` of `n` value-initialized elements.
Running time: Linear in the number of elements inserted (`n`)

```
vector(size_type n, const T& value, const Allocator& = Allocator());
```

Constructs a `vector` of size `n`, with optional allocator, and initializes elements to `value`.
Running time: Linear in the number of elements inserted (`n`)

```
template <class 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.
Running time: Linear in the number of elements inserted

```
vector(const vector<T,Allocator>& x);
```

Copy constructor.
Running time: Linear in the size of `x`

```
vector(vector<T,Allocator>&& x);
```

Move constructor.
Running time: Constant when the allocators are the same, linear otherwise

```
vector(const vector<T,Allocator>& x, const Allocator&);
```

Copy constructor using specified allocator.
Running time: Linear in the size of `x`

```
vector(vector<T,Allocator>&&, const Allocator&);
```

Move constructor using specified allocator.
Running time: Constant when the allocators are the same, linear otherwise

```
vector(initializer_list<T>, const Allocator& = Allocator());
```

Initializer list constructor.
Running time: Linear in the size of the initializer list

```
~vector();
```

Destructor.
Running time: Linear (destructor is called on every element in the `vector`)
Invalidates Iterators and References? Yes

```
vector<T,Allocator>& operator=(const vector<T,Allocator>& x);
```

Copy assignment operator.
Running time: Linear in *S* plus the size of `x`
Invalidates Iterators and References? Yes

```
vector<T,Allocator>& operator=(vector<T,Allocator>&& x);
```

Move assignment operator.
Running time: Constant when the allocators are the same, linear otherwise
Invalidates Iterators and References? Yes

```
vector<T,Allocator>& operator=(initializer_list<T> il);
```

Assignment operator accepting an initializer list as right-hand side.
Running time: Linear in *S* plus the number of elements in `il`
Invalidates Iterators and References? Yes

```
template <class InputIterator>
    void assign(InputIterator first, InputIterator last);
```

Removes all the current elements and inserts all the elements from `first` to `last`.
Running time: Linear in *S* plus the number of elements inserted
Invalidates Iterators and References? Yes

```
void assign(size_type n, const T& u);
```

Removes all the current elements and inserts `n` elements of value `u`.
Running time: Linear in S plus `n`
Invalidates Iterators and References? Yes

```
void assign(initializer_list<T> il);
```

Removes all the current elements and inserts all elements of the initializer list.
Running time: Linear in *S* plus the number of elements in `il`
Invalidates Iterators and References? Yes

```
void swap(vector<T, Allocator>&);
```

Swaps the contents of the two `vector`s.
Running time: Usually constant (just swaps internal pointers), but not required by the standard
Invalidates Iterators and References? No (but the iterators and references now refer to elements in a different container)

Adding and Deleting Elements

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

```
void push_back(const T& x);
void push_back(T&& x);
```

Inserts element `x` at the end of the `vector` by copying or moving it.
Running time: Amortized constant
Invalidates Iterators and References? Yes (if reallocation occurs)

```
template <class... Args> void emplace_back(Args&&... args);
```

Creates an element in-place at the end of the `vector`.
Running time: Amortized constant
Invalidates Iterators and References? Yes (if reallocation occurs)

```
template <class... Args> iterator emplace(const_iterator position,
                                          Args&&... args);
```

Creates an element in-place at `position`. Returns an iterator referring to the element inserted.
Running time: Amortized constant
Invalidates Iterators and References? Yes (all if reallocation occurs; otherwise only those referring to elements at or past the insertion point)

```
void pop_back();
```

Removes the last element in the `vector`.
Running time: Constant
Invalidates Iterators and References? Only iterators and references referring to that element

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
```

Inserts the element `x` (by copying or moving it) before the element at `position`, shifting all subsequent elements to make room. Returns an iterator referring to the element inserted.
Running time: Amortized constant at the end; linear elsewhere
Invalidates Iterators and References? Yes (all if reallocation occurs; otherwise only those referring to elements at or past the insertion point)

```
iterator insert(const_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. Returns an iterator referring to the first element inserted.
Running time: Amortized constant at the end; linear in *S* plus `n` elsewhere
Invalidates Iterators and References? Yes (all if reallocation occurs; otherwise only those referring to elements at or past the insertion point)

```
template <class InputIterator>
    iterator insert(const_iterator position,
                    InputIterator first, InputIterator last);
```

Inserts all elements from `first` to `last` before the element at `position`.
Running time: Amortized constant at the end; linear in *S* plus the number of elements inserted elsewhere**Invalidates Iterators and References?** Yes (all if reallocation occurs; otherwise only those referring to elements at or past the insertion point)

```
iterator insert(const_iterator position, initializer_list<T> il);
```

Inserts all elements from the initializer list before the element at `position`.
Running time: Amortized constant at the end; linear in *S* plus the number of elements inserted elsewhere
Invalidates Iterators and References? Yes (all if reallocation occurs; otherwise only those referring to elements at or past the insertion point)

```
iterator erase(const_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.
Running time: Constant at the end; linear elsewhere
Invalidates Iterators and References? Invalidates all iterators and references at or past `position`

```
iterator erase(const_iterator first, const_iterator last);
```

Removes the elements from `first` to `last`. Returns an iterator referring to the element following the ones that were erased.
Running time: Constant at the end; linear elsewhere
Invalidates Iterators and References? Invalidates all iterators and references at or past `first`

```
void clear() noexcept;
```

Erases all elements in the `vector`.
Running time: Linear
Invalidates Iterators and References? 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.

```
reference operator[](size_type n);
const_reference operator[](size_type n) const;
```

Array syntax for element access. Does not perform bounds checking.
Running time: Constant
Invalidates Iterators and References? 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.
Running time: Constant
Invalidates Iterators and References? No

```
reference front();
const_reference front() const;
```

Returns a `reference` to the first element. Undefined if there are no elements.
Running time: Constant
Invalidates Iterators and References? No

```
reference back();
const_reference back() const;
```

Returns a `reference` to the last element. Undefined if there are no elements.
Running time: Constant
Invalidates Iterators and References? No

```
T* data() noexcept;
const T* data() const noexcept;
```

Returns a pointer to the data.
Running time: Constant
Invalidates Iterators and References? No

Retrieving and Setting Size and Capacity

```
size_type size() const noexcept;
```

The number of elements in the `vector`.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
size_type max_size() const noexcept;
```

The maximum number of elements the `vector` could hold. Not usually a very useful method, because the number is likely to be quite large.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
void resize(size_type sz);
void resize(size_type sz, const T& c);
```

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.
Running time: Linear
Invalidates Iterators and References? Yes (if reallocation occurs)

```
size_type capacity() const noexcept;
```

The number of elements the `vector` could hold without a reallocation.
Running time: Usually constant (but unspecified)
Invalidates Iterators and References? No

```
bool empty() const noexcept;
```

Returns `true` if the `vector` currently has no elements; `false` otherwise.
Running time: Constant
Invalidates Iterators and References? No

```
void reserve(size_type n);
```

Changes the capacity of the `vector` to `n`. Does not change the size of the `vector`. Throws `length_error` if `n > max_size()`.
Running time: Linear
Invalidates Iterators and References? Yes (if reallocation occurs)

```
void shrink_to_fit();
```

Non-binding request to reduce the capacity of the `vector` to its size.
Running time: Constant
Invalidates Iterators and References? No

The vector<bool> Specialization

The partial specialization of `vector<bool>` provides almost the same 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 `reference`s (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:

```
void flip() noexcept;
```

If called on the container, complements all the elements. If called on a reference object, complements that element.
Running time: Linear on container; constant on reference object
Invalidates Iterators and References? No

array

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

Iterator

The `array` provides random-access iteration.

Template Definition

```
template <class T, size_t N > struct array;
```

`T` is the element type to be stored in the `array`, and `N` is the fixed number of elements in your `array`.

Assignment Methods

```
void swap(array<T, N>&);
```

Swaps the contents of the two `array`s.
Running time: Linear in `N`
Invalidates Iterators and References? No (but the iterators and references now refer to elements in a different container)

Accessing Elements

`array`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 array`, 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.

```
reference operator[](size_type n);
const_reference operator[](size_type n) const;
```

Array syntax for element access. Does not perform bounds checking.
Running time: Constant
Invalidates Iterators and References? 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.
Running time: Constant
Invalidates Iterators and References? No

```
reference front();
const_reference front() const;
```

Returns a `reference` to the first element. Undefined if there are no elements.
Running time: Constant
Invalidates Iterators and References? No

```
reference back();
const_reference back() const;
```

Returns a `reference` to the last element. Undefined if there are no elements.
Running time: Constant
Invalidates Iterators and References? No

```
T* data() noexcept;
const T* data() const noexcept;
```

Returns a pointer to the data.
Running time: Constant
Invalidates Iterators and References? No

Retrieving Size

```
constexpr size_type size() noexcept;
```

The number of elements in the `array`.
Running time: Constant
Invalidates Iterators and References? No

```
constexpr size_type max_size() noexcept;
```

The maximum number of elements the `array` could hold. Not usually a very useful method, because the number is likely to be quite large.
Running time: Constant
Invalidates Iterators and References? No

```
constexpr bool empty() noexcept;
```

Returns `true` if the `array` has no elements; `false` otherwise.
Running time: Constant
Invalidates Iterators and References? No

array Operations

```
void fill(const T& u);
```

Sets each element of the `array` to the value `u`.
Running time: Linear
Invalidates Iterators and References? 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 <class T, class Allocator = allocator<T> > class deque;
```

`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

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

Default constructor; constructs a `deque` of size 0, optionally with the specified allocator.
Running time: Constant

```
explicit deque(size_type n);
```

Constructs a `deque` of `n` value-initialized elements.
Running time: Linear in the number of elements inserted (`n`)

```
deque(size_type n, const T& value,const Allocator& = Allocator());
```

Constructs a `deque` of size `n`, with optional allocator, and initializes elements to `value`.
Running time: Linear in the number of elements inserted (`n`)

```
template <class InputIterator>
    deque(InputIterator first, InputIterator last,
          const Allocator& = Allocator());
```

Constructs a `deque`, with optional allocator, and inserts the elements from `first` to `last`. It's a method template in order to work on any iterator.
Running time: Linear in the number of elements inserted

```
deque(const deque<T,Allocator>& x);
```

Copy constructor.
Running time: Linear in the size of `x`

```
deque(deque<T,Allocator>&& x);
```

Move constructor.
Running time: Constant when the allocators are the same, linear otherwise

```
deque(const deque<T,Allocator>& x, const Allocator&);
```

Copy constructor using specified allocator.
Running time: Linear in the size of `x`

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

Move constructor using specified allocator.
Running time: Constant when the allocators are the same, linear otherwise

```
deque(initializer_list<T>, const Allocator& = Allocator());
```

Initializer list constructor.
Running time: Linear in the size of the initializer list

```
~deque();
```

Destructor.
Running time: Linear (destructor is called on every element in the `deque`)
Invalidates Iterators and References? Yes

```
deque<T,Allocator>& operator=(const deque<T,Allocator>& x);
```

Copy assignment operator.
Running time: Linear in *S* plus the size of `x`
Invalidates Iterators and References? Yes

```
deque<T,Allocator>& operator=(deque<T,Allocator>&& x);
```

Move assignment operator.
Running time: Constant when the allocators are the same, linear otherwise
Invalidates Iterators and References? Yes

```
deque<T,Allocator>& operator=(initializer_list<T> il);
```

Assignment operator accepting an initializer list as right-hand side.
Running time: Linear in *S* plus the number of elements in `il`
Invalidates Iterators and References? Yes

```
template <class InputIterator>
    void assign(InputIterator first, InputIterator last);
```

Removes all the current elements and inserts all the elements from `first` to `last`.
Running time: Linear in *S* plus number of elements inserted
Invalidates Iterators and References? Yes

```
void assign(size_type n, const T& t);
```

Removes all the current elements and inserts `n` elements of value `t`.
Running time: Linear in *S* plus `n`
Invalidates Iterators and References? Yes

```
void assign(initializer_list<T> il);
```

Removes all the current elements and inserts all elements of the initializer list.
Running time: Linear in *S* plus the number of elements in `il`
Invalidates Iterators and References? Yes

```
void swap(deque<T, Allocator>&);
```

Swaps the contents of the two `deque`s.
Running time: Usually constant (just swaps internal pointers), but not required by the standard
Invalidates Iterators and References? 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.

```
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
```

Inserts element `x` at the beginning or end of the `deque` by copying or moving it.
Running time: Constant
Invalidates Iterators and References? Iterators: yes, References: no

```
template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
```

Creates an element in-place at the beginning or end of the `deque`.
Running time: Constant
Invalidates Iterators and References? Iterators: yes, References: no

```
template <class... Args> iterator emplace(const_iterator position,
                                          Args&&... args);
```

Creates an element in-place at `position`. Returns an iterator referring to the element inserted.
Running time: Constant
Invalidates Iterators and References? Iterators: yes, References: no

```
void pop_front();
void pop_back();
```

Removes the first or last element in the `deque`.
Running time: Constant
Invalidates Iterators and References? Only iterators and references referring to that element

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
```

Inserts the element `x` (by copying or moving it) before the element at `position`, shifting all subsequent elements to make room. Returns an iterator referring to the element inserted.
Running time: Linear in the middle; constant at beginning or end
Invalidates Iterators and References? Yes, unless the element is added to the front or back; then, only iterators are invalidated

```
iterator insert(const_iterator position, size_type n, const T& x);
```

Inserts `n` copies of `x` before the element at `position`.
Running time: Linear in *S* plus n when inserting in the the middle; constant at beginning or end
Invalidates Iterators and References? Yes, unless the elements are added to the front or back; then, only iterators are invalidated

```
template <class InputIterator>
    iterator insert(const_iterator position,
                    InputIterator first, InputIterator last);
```

Inserts all elements from `first` to `last` before the element at `position`.
Running time: Linear in *S* plus number of elements inserted when inserting in the middle; constant at beginning or end
Invalidates Iterators and References? Yes, unless the elements are added to the front or back; then, only iterators are invalidated

```
iterator insert(const_iterator position, initializer_list<T> il);
```

Inserts all elements from the initializer list before the element at `position`.
Running time: Linear in *S* plus number of elements inserted when inserting in the middle; constant at beginning or end
Invalidates Iterators and References? Yes, unless the elements are added to the front or back; then, only iterators are invalidated

```
iterator erase(const_iterator position);
```

Removes the element at `position`. Returns an iterator referring to the element following the one that was erased.
Running time: Linear in the middle; constant at beginning or end
Invalidates Iterators and References? Yes, unless the erased element is at the front or back; then, only iterators and references to that element are invalidated

```
iterator erase(const_iterator first, const_iterator last);
```

Removes the elements from `first` to `last`. Returns an iterator referring to the element following the ones that were erased.
Running time: Linear in the middle; constant at beginning or end
Invalidates Iterators and References? Yes, unless the erased elements are at the front or back; then only iterators and references to that element are invalidated

```
void clear() noexcept;
```

Erases all elements in the `deque`.
Running time: Linear
Invalidates Iterators and References? Yes

Accessing Elements

`deque`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 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.

```
reference operator[](size_type n);
const_reference operator[](size_type n) const;
```

Array syntax for element access. Does not perform bounds checking.
Running time: Constant
Invalidates Iterators and References? 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.
Running time: Constant
Invalidates Iterators and References? No

```
reference front();
const_reference front() const;
```

Returns a `reference` to the first element. Undefined if there are no elements.
Running time: Constant
Invalidates Iterators and References? No

```
reference back();
const_reference back() const;
```

Returns a `reference` to the last element. Undefined if there are no elements.
Running time: Constant
Invalidates Iterators and References? No

Retrieving and Setting Size

```
size_type size() const noexcept;
```

The number of elements in the `deque`.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
size_type max_size() const noexcept;
```

The maximum number of elements the `deque` could hold. Not usually a very useful method, because the number is likely to be quite large.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
void resize(size_type sz);
void resize(size_type sz, const T& c);
```

Changes the number of elements in the `deque` (the size) to `sz`, creating new ones with the default constructor if required.
Running time: Linear
Invalidates Iterators and References? Yes

```
bool empty() const noexcept;
```

Returns `true` if the `deque` currently has no elements; `false` otherwise.
Running time: Constant
Invalidates Iterators and References? No

```
void shrink_to_fit();
```

Non-binding request to reduce memory use.
Running time: Constant
Invalidates Iterators and References? 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 <class T, class Allocator = allocator<T> > class list;
```

`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

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

Default constructor; constructs a `list` of size 0, optionally with the specified allocator.
Running time: Constant

```
explicit list(size_type n);
```

Constructs a `list` of `n` value-initialized elements.
Running time: Linear in the number of elements inserted (`n`)

```
list(size_type n, const T& value, const Allocator& = Allocator());
```

Constructs a `list` of size `n`, with optional allocator, and initializes elements to `value`.
Running time: Linear in the number of elements inserted (`n`)

```
template <class InputIterator>
    list(InputIterator first, InputIterator last,
         const Allocator& = Allocator());
```

Constructs a `list`, with optional allocator, and inserts the elements from `first` to `last`. It's a method template in order to work on any iterator.
Running time: Linear in the number of elements inserted

```
list(const list<T,Allocator>& x);
```

Copy constructor.
Running time: Linear in the size of `x`

```
list(list<T,Allocator>&& x);
```

Move constructor.
Running time: Constant when the allocators are the same, linear otherwise

```
list(const list<T,Allocator>& x, const Allocator&);
```

Copy constructor using specified allocator.
Running time: Linear in the size of `x`

```
list(list<T,Allocator>&&, const Allocator&);
```

Move constructor using specified allocator.
Running time: Constant when the allocators are the same, linear otherwise

```
list(initializer_list<T>, const Allocator& = Allocator());
```

Initializer list constructor.
Running time: Linear in the size of the initializer list

```
~list();
```

Destructor.
Running time: Linear (destructor is called on every element in the `list`)
Invalidates Iterators and References? Yes

```
list<T,Allocator>& operator=(const list<T,Allocator>& x);
```

Copy assignment operator.
Running time: Linear in *S* plus the size of `x`
Invalidates Iterators and References? Yes

```
list<T,Allocator>& operator=(list<T,Allocator>&& x);
```

Move assignment operator.
Running time: Constant when the allocators are the same, linear otherwise
Invalidates Iterators and References? Yes

```
list<T,Allocator>& operator=(initializer_list<T> il);
```

Assignment operator accepting an initializer list as right-hand side.
Running time: Linear in *S* plus the number of elements in `il`
Invalidates Iterators and References? Yes

```
template <class InputIterator>
    void assign(InputIterator first, InputIterator last);
```

Removes all the current elements and inserts all the elements from `first` to `last`.
Running time: Linear in *S* plus number of elements inserted
Invalidates Iterators and References? Yes

```
void assign(size_type n, const T& t);
```

Removes all the current elements and inserts `n` elements of value `t`.
Running time: Linear in *S* plus `n`
Invalidates Iterators and References? Yes

```
void assign(initializer_list<T> il);
```

Removes all the current elements and inserts all elements of the initializer list.
Running time: Linear in *S* plus the number of elements in `il`
Invalidates Iterators and References? Yes

```
void swap(list<T,Allocator>&);
```

Swaps the contents of the two `list`s.
Running time: Usually constant (just swaps internal pointers), but not required by the standard
Invalidates Iterators and References? 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.

```
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
```

Inserts element `x` at the beginning or end of the `list` by copying or moving it.
Running time: Constant
Invalidates Iterators and References? No

```
template <class... Args> void emplace_front(Args&&... args);
template <class... Args> void emplace_back(Args&&... args);
```

Creates an element in-place at the beginning or end of the `list`.
Running time: Constant
Invalidates Iterators and References? No

```
template <class... Args> iterator emplace(const_iterator position,
                                          Args&&... args);
```

Creates an element in-place at `position`. Returns an iterator referring to the element inserted.
Running time: Constant
Invalidates Iterators and References? No

```
void pop_front();
void pop_back();
```

Removes the first or last element in the `list`.
Running time: Constant
Invalidates Iterators and References? Only those referring to the erased element

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
```

Inserts the element `x` (by copying or moving it) before the element at `position`. Returns an iterator referring to the element inserted.
Running time: Constant
Invalidates Iterators and References? No

```
iterator insert(const_iterator position, size_type n, const T& x);
```

Inserts `n` copies of `x` before the element at `position`.
Running time: Constant in *S*; Linear in `n`
Invalidates Iterators and References? No

```
template <class InputIterator>
    iterator insert(const_iterator position,
                    InputIterator first, InputIterator last);
```

Inserts all elements from `first` to `last` before the element at `position`.
Running time: Constant in *S*; linear in number of elements inserted
Invalidates Iterators and References? No

```
iterator insert(const_iterator position, initializer_list<T> il);
```

Inserts all elements from the initializer list before the element at `position`.
Running time: Constant in *S*; linear in number of elements inserted
Invalidates Iterators and References? No

```
iterator erase(const_iterator position);
```

Removes the element at `position`. Returns an iterator referring to the element following the one that was erased.
Running time: Constant
Invalidates Iterators and References? Only those referring to the erased element

```
iterator erase(const_iterator first, const_iterator last);
```

Removes the elements from `first` to `last`. Returns an iterator referring to the element following the ones that were erased.
Running time: Constant in *S*; linear in number of elements erased
Invalidates Iterators and References? Only those referring to the erased elements

```
void clear() noexcept;
```

Erases all elements in the `list`.
Running time: Linear
Invalidates Iterators and References? Yes

Accessing Elements

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

```
reference front();
const_reference front() const;
```

Returns a `reference` to the first element. Undefined if there are no elements.
Running time: Constant
Invalidates Iterators and References? No

```
reference back();
const_reference back() const;
```

Returns a `reference` to the last element. Undefined if there are no elements.
Running time: Constant
Invalidates Iterators and References? No

Retrieving and Setting Size

```
size_type size() const noexcept;
```

The number of elements in the `list`.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
size_type max_size() const noexcept;
```

The maximum number of elements the `list` could hold. Not usually a very useful method, because the number is likely to be quite large.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
void resize(size_type sz);
void resize(size_type sz, const T& c);
```

Changes the number of elements in the `list` (the size) to `sz`, creating new ones with the default constructor if required.
Running time: Linear
Invalidates Iterators and References? Yes

```
bool empty() const noexcept;
```

Returns `true` if the `list` currently has no elements; `false` otherwise.
Running time: Constant
Invalidates Iterators and References? No

```
void shrink_to_fit();
```

Non-binding request to reduce memory use.
Running time: Constant
Invalidates Iterators and References? 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.

```
void splice(const_iterator position, list<T,Allocator>& x);
void splice(const_iterator position, list<T,Allocator>&& x);
```

Inserts the `list` `x` into `position` by copying or moving. Destroys `x`.
Running time: Constant
Invalidates Iterators and References? Invalidates iterators and references to `x`; Does not invalidate those to the list on which the method is called

```
void splice(const_iterator position, list<T,Allocator>& x, const_iterator i);
void splice(const_iterator position, list<T,Allocator>&& x, const_iterator i);
```

Inserts element `i` from `list` `x` into `position` by copying or moving. Removes element `i` from `x`.
Running time: Constant
Invalidates Iterators and References? Invalidates iterators and references only to the element referred to by `i`

```
void splice(const_iterator position, list<T,Allocator>& x,
            const_iterator first, const_iterator last);
void splice(const_iterator position, list<T,Allocator>&& x,
            const_iterator first, const_iterator last);
```

Assumes that `first` and `last` are iterators into `x`. Inserts the specified range from `x` into `position` by copying or moving. Removes the range from `x`.
Running time: Constant
Invalidates Iterators and References? 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`.
Running time: Linear
Invalidates Iterators and References? 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`.
Running time: Linear
Invalidates Iterators and References? Invalidates iterators and references to the erased elements

```
void merge(list<T,Allocator>& x);
void merge(list<T,Allocator>&& x);
template <class Compare> void merge(list<T,Allocator>& x, Compare comp);
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`.
Running time: Linear
Invalidates Iterators and References? Invalidates 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.
Running time: Linear logarithmic
Invalidates Iterators and References? No

```
void reverse() noexcept;
```

Reverses the order of the elements in the `list`.
Running time: Linear
Invalidates Iterators and References? No

forward_list

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

Iterator

The `forward_list` provides only forward iteration.

The `forward_list` does not provide `rbegin()`, `rend()`, `crbegin()`, and `crend()`, but it does provide the following extra methods:

```
iterator before_begin() noexcept;
const_iterator before_begin() const noexcept;
const_iterator cbefore_begin() const noexcept;
```

Returns: An `iterator` (or `const_iterator`) referring to the "before-the-first" element in the container. Thus, incrementing the returned iterator points to the first element of the `forward_list`.
Running time: Constant

Template Definition

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

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

Constructors, Destructors, and Assignment Methods

```
explicit forward_list(const Allocator& = Allocator());
```

Default constructor; constructs a `forward_list` of size 0, optionally with the specified allocator.
Running time: Constant

```
explicit forward_list(size_type n);
```

Constructs a `forward_list` of `n` value-initialized elements.
Running time: Linear in the number of elements inserted (`n`)

```
forward_list(size_type n, const T& value, const Allocator& = Allocator());
```

Constructs a `forward_list` of size `n`, with optional allocator, and initializes elements to `value`.
Running time: Linear in the number of elements inserted (`n`)

```
template <class InputIterator>
    forward_list(InputIterator first, InputIterator last,
         const Allocator& = Allocator());
```

Constructs a `forward_list`, with optional allocator, and inserts the elements from `first` to `last`. It's a method template in order to work on any iterator.
Running time: Linear in the number of elements inserted

```
forward_list(const forward_list<T,Allocator>& x);
```

Copy constructor.
Running time: Linear in the size of `x`

```
forward_list(forward_list<T,Allocator>&& x);
```

Move constructor.
Running time: Constant when the allocators are the same, linear otherwise

```
forward_list(const forward_list<T,Allocator>& x, const Allocator&);
```

Copy constructor using specified allocator.
Running time: Linear in the size of `x`

```
forward_list(forward_list<T,Allocator>&&, const Allocator&);
```

Move constructor using specified allocator.
Running time: Constant when the allocators are the same, linear otherwise

```
forward_list(initializer_list<T>, const Allocator& = Allocator());
```

Initializer list constructor.
Running time: Linear in the size of the initializer list

```
~forward_list();
```

Destructor.
Running time: Linear (destructor is called on every element in the `forward_list`)
Invalidates Iterators and References? Yes

```
forward_list<T,Allocator>& operator=(const forward_list<T,Allocator>& x);
```

Copy assignment operator.
Running time: Linear in *S* plus the size of `x`
Invalidates Iterators and References? Yes

```
forward_list<T,Allocator>& operator=(forward_list<T,Allocator>&& x);
```

Move assignment operator.
Running time: Constant when the allocators are the same, linear otherwise
Invalidates Iterators and References? Yes

```
forward_list<T,Allocator>& operator=(initializer_list<T> il);
```

Assignment operator accepting an initializer list as right-hand side.
Running time: Linear in *S* plus the number of elements in `il`
Invalidates Iterators and References? Yes

```
template <class InputIterator>
    void assign(InputIterator first, InputIterator last);
```

Removes all the current elements and inserts all the elements from `first` to `last`.
Running time: Linear in *S* plus number of elements inserted
Invalidates Iterators and References? Yes

```
void assign(size_type n, const T& t);
```

Removes all the current elements and inserts `n` elements of value `t`.
Running time: Linear in *S* plus `n`
Invalidates Iterators and References? Yes

```
void assign(initializer_list<T> il);
```

Removes all the current elements and inserts all elements of the initializer list.
Running time: Linear in *S* plus the number of elements in `il`
Invalidates Iterators and References? Yes

```
void swap(forward_list<T,Allocator>&);
```

Swaps the contents of the two `forward_list`s.
Running time: Usually constant (just swaps internal pointers), but not required by the standard
Invalidates Iterators and References? No (but the iterators and references now refer to elements in a different container)

Adding and Deleting Elements

```
void push_front(const T& x);
void push_front(T&& x);
```

Inserts element `x` at the beginning of the `forward_list` by copying or moving it.
Running time: Constant
Invalidates Iterators and References? No

```
template <class... Args> void emplace_front(Args&&... args);
```

Creates an element in-place at the beginning of the `forward_list`.
Running time: Constant
Invalidates Iterators and References? No

```
template <class... Args> iterator emplace_after(const_iterator position,
                                                Args&&... args);
```

Creates an element in-place after `position`. Returns an iterator referring to the element inserted.
Running time: Constant
Invalidates Iterators and References? No

```
void pop_front();
```

Removes the first element in the `forward_list`.
Running time: Constant
Invalidates Iterators and References? Only those referring to the erased element

```
iterator insert_after(const_iterator position, const T& x);
iterator insert_after(const_iterator position, T&& x);
```

Inserts the element `x` (by copying or moving it) after the element at `position`. Returns an iterator referring to the element inserted.
Running time: Constant
Invalidates Iterators and References? No

```
iterator insert_after(const_iterator position, size_type n, const T& x);
```

Inserts `n` copies of `x` after the element at `position`.
Running time: Constant in *S*; Linear in `n`
Invalidates Iterators and References? No

```
template <class InputIterator>
    iterator insert_after(const_iterator position,
                          InputIterator first, InputIterator last);
```

Inserts all elements from `first` to `last` after the element at `position`.
Running time: Constant in *S*; linear in number of elements inserted
Invalidates Iterators and References? No

```
iterator insert_after(const_iterator position, initializer_list<T> il);
```

Inserts all elements from the initializer list after the element at `position`.
Running time: Constant in *S*; linear in number of elements inserted
Invalidates Iterators and References? No

```
iterator erase_after(const_iterator position);
```

Removes the element after `position`. Returns an iterator referring to the element following the one that was erased.
Running time: Constant
Invalidates Iterators and References? Only those referring to the erased element

```
iterator erase_after(const_iterator position, iterator last);
```

Removes the elements in the range (`position, last`). Returns `last`.
Running time: Constant in *S*; linear in number of elements erased
Invalidates Iterators and References? Only those referring to the erased elements

```
void clear() noexcept;
```

Erases all elements in the `forward_list`.
Running time: Linear
Invalidates Iterators and References? Yes

Accessing Elements

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

```
reference front();
const_reference front() const;
```

Returns a `reference` to the first element. Undefined if there are no elements.
Running time: Constant
Invalidates Iterators and References? No

Retrieving and Setting Size

```
size_type max_size() const noexcept;
```

The maximum number of elements the `forward_list` could hold. Not usually a very useful method, because the number is likely to be quite large.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
void resize(size_type sz);
void resize(size_type sz, const value_type& c);
```

Changes the number of elements in the `forward_list` to `sz`, creating new ones with the default constructor if required.
Running time: Linear
Invalidates Iterators and References? Yes

```
bool empty() const noexcept;
```

Returns `true` if the `forward_list` currently has no elements; `false` otherwise.
Running time: Constant
Invalidates Iterators and References? No

forward_list Operations

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

```
void splice_after(const_iterator position, forward_list<T,Allocator>& x);
void splice_after(const_iterator position, forward_list<T,Allocator>&& x);
```

Inserts the `forward_list` `x` after `position` by copying or moving. Destroys `x`.
Running time: Constant
Invalidates Iterators and References? Invalidates iterators and references to `x`; Does not invalidate those to the list on which the method is called

```
void splice_after(const_iterator position, forward_list<T,Allocator>& x,
                  const_iterator i);
void splice_after(const_iterator position, forward_list<T,Allocator>&& x,
                  const_iterator i);
```

Inserts element `i` from `forward_list` `x` after `position` by copying or moving. Removes element `i` from `x`.
Running time: Constant
Invalidates Iterators and References? Invalidates iterators and references only to the element referred to by `i`

```
void splice_after(const_iterator position, forward_list<T,Allocator>& x,
                  const_iterator first, const_iterator last);
void splice_after(const_iterator position, forward_list<T,Allocator>&& x,
                  const_iterator first, const_iterator last);
```

Assumes that `first` and `last` are iterators into `x`. Inserts the specified range from `x` after `position` by copying or moving. Removes the range from `x`.
Running time: Constant
Invalidates Iterators and References? 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 `forward_list` equal to `value` or for which `pred` is `true`.
Running time: Linear
Invalidates Iterators and References? Invalidates iterators and references to the erased elements

```
void unique();
template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);
```

Removes duplicate consecutive elements from the `forward_list`. Duplicates are checked with `operator==` or `binary_pred`.
Running time: Linear
Invalidates Iterators and References? Invalidates iterators and references to the erased elements

```
void merge(forward_list<T,Allocator>& x);
void merge(forward_list<T,Allocator>&& x);
template <class Compare> void merge(forward_list<T,Allocator>& x, Compare comp);
template <class Compare> void merge(forward_list<T,Allocator>&& x, Compare comp);
```

Merges `x` into the `forward_list` on which the method is called. Both `forward_list`s must be sorted to start. `x` is empty after the merge. Compares elements with `operator<` or `comp`.
Running time: Linear
Invalidates Iterators and References? Invalidates all iterators and references to elements in `x`

```
void sort();
template <class Compare> void sort(Compare comp);
```

Performs stable sort on elements in the `forward_list`, using `operator<` or the specified `comp` to order elements.
Running time: Linear logarithmic
Invalidates Iterators and References? No

```
void reverse() noexcept;
```

Reverses the order of the elements in the `forward_list`.
Running time: Linear
Invalidates Iterators and References? 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 <class T, class Container = deque<T> > class stack;
```

`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 a couple of constructors. It provides no destructor, copy constructor, move constructor, or assignment operator because its only data member, the underlying container, handles all of that.

```
explicit stack(const Container&);
explicit stack(Container&& = Container());
template <class Alloc> explicit stack(const Alloc&);
template <class Alloc> stack(const Container&, const Alloc&);
template <class Alloc> stack(Container&&, const Alloc&);
template <class Alloc> stack(const stack&, const Alloc&);
template <class Alloc> stack(stack&&, const Alloc&);
```

Constructs a `stack` using the specified container to store its elements, and the specified allocator.
Running time: Constant

```
void swap(stack<T, Container>&);
```

Swaps the contents of the two `stack`s.
Running time: Usually constant (just swaps internal pointers), but not required by the standard

Adding and Deleting Elements

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

```
void push(const value_type& x);
void push(value_type&& x);
```

Inserts element `x` at the top of the `stack`, by copying or moving `x`.
Running time: Constant

```
template <class... Args> void emplace(Args&&... args);
```

Creates an element in-place at the top of the `stack`.
Running time: Constant

```
void pop();
```

Removes the top element from the `stack`.
Running time: Constant

Accessing Elements

```
reference top();
const_reference top() const;
```

Retrieves (but does not remove) the top value of the `stack`. Returns a `const reference` if called on a `const` object; otherwise returns a `reference`.
Running time: Constant

Retrieving Size

```
size_type size() const;
```

The number of elements in the `stack`.
Running time: Usually constant, but not required to be by the standard

```
bool empty() const;
```

Returns `true` if the `stack` currently has no elements; `false` otherwise.
Running time: Constant

queue

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

Template Definition

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

`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`. The `vector` container 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 a couple of constructors. It provides no destructor, copy constructor, move constructor, or assignment operator because its only data member, the underlying container, handles all of that.

```
explicit queue(const Container&);
explicit queue(Container&& = Container());
template <class Alloc> explicit queue(const Alloc&);
template <class Alloc> queue(const Container&, const Alloc&);
template <class Alloc> queue(Container&&, const Alloc&);
template <class Alloc> queue(const queue&, const Alloc&);
template <class Alloc> queue(queue&&, const Alloc&);
```

Constructs a `queue` using the specified container to store its elements, and the specified allocator.
Running time: Constant

```
void swap(queue<T, Container>&);
```

Swaps the contents of the two `queue`s.
Running time: Usually constant (just swaps internal pointers), but not required by the standard

Adding and Deleting Elements

`queue`s provide two ways to add elements and one way to remove them.

```
void push(const value_type& x);
void push(value_type&& x);
```

Inserts element `x` at the end of the `queue`, by copying or moving `x`.
Running time: Constant

```
template <class... Args> void emplace(Args&&... args);
```

Creates an element in-place at the end of the `queue`.
Running time: Constant

```
void pop();
```

Removes the front element from the `queue`.
Running time: Constant

Accessing Elements

```
reference front();
const_reference front();
reference back();
const_reference back();
```

Retrieves (but does not remove) the front or back value of the `queue`. Returns a `const reference` if called on a `const` object; otherwise returns a `reference`.
Running time: Constant

Retrieving Size

```
size_type size() const;
```

The number of elements in the `queue`.
Running time: Usually constant, but not required to be by the standard

```
bool empty() const;
```

Returns `true` if the `queue` currently has no elements; `false` otherwise.
Running time: Constant

priority_queue

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

Template Definition

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

`T` is the element type to be stored in the `priority_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`. The `list` container 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, move constructor, or assignment operator because its underlying container handles all of that.

```
priority_queue(const Compare& x, const Container&);
explicit priority_queue(const Compare& x = Compare(),
                        Container&& = Container());
template <class Alloc> explicit priority_queue(const Alloc&);
template <class Alloc> priority_queue(const Compare&, const Alloc&);
template <class Alloc> priority_queue(const Compare&,const Container&,
                                      const Alloc&);
template <class Alloc> priority_queue(const Compare&,Container&&,
                                      const Alloc&);
template <class Alloc> priority_queue(const priority_queue&, const Alloc&);
template <class Alloc> priority_queue(priority_queue&&, const Alloc&);
```

Constructs a `priority_queue` using the specified comparison callback, specified allocator, and the specified container to store its elements.
Running time: Constant

```
template <class InputIterator>
    priority_queue(InputIterator first, InputIterator last,
                   const Compare& x, const Container&);

template <class InputIterator>
    priority_queue(InputIterator first, InputIterator last,
                   const Compare& x = Compare(), Container&& = Container());
```

Constructs a `priority_queue` using the specified comparison function, and the specified container to store its elements. Inserts elements from `first` to `last` into the container.
Running time: Linear in the number of elements in the container and the range `first` to `last`.

```
void swap(priority_queue<T, Container, Compare>&);
```

Swaps the contents of the two `priority_queue`s.
Running time: Usually constant (just swaps internal pointers), but not required by the standard

Adding and Deleting Elements

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

```
void push(const value_type& x);
void push(value_type&& x);
```

Inserts element `x` in its priority order in the `priority_queue`, by copying or moving `x`.
Running time: Logarithmic

```
template <class... Args> void emplace(Args&&... args);
```

Creates an element in-place in the `priority_queue`.
Running time: Logarithmic

```
void pop();
```

Removes the element with the highest priority from the `priority_queue`.
Running time: Logarithmic

Accessing Elements

```
const_reference top() const;
```

Retrieves (but does not remove) a `const` reference to the element with the highest priority in the `priority_queue`.
Running time: Constant

Retrieving Size

```
size_type size() const;
```

The number of elements in the `priority_queue`.
Running time: 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.
Running time: Constant

Associative Containers

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

Associative Container typedefs

Associative containers add the following `typedef`s to the common container `typedef`s.

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 as 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 <class Key, class T, class Compare = less<Key>,
          class 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

```
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.
Running time: Constant

```
explicit map(const Allocator&);
```

Constructor; constructs a `map` of size `0`, with the specified allocator.
Running time: Constant

```
template <class 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.
Running time: Linear logarithmic in the number of elements inserted; linear if the inserted element range is already sorted according to `comp`

```
map(const map<Key,T,Compare,Allocator>& x);
```

Copy constructor.
Running time: Linear in the size of `x`

```
map(map<Key,T,Compare,Allocator>&& x);
```

Move constructor.
Running time: Constant when the allocators are the same, linear otherwise

```
map(const map<Key,T,Compare,Allocator>& x, const Allocator&);
```

Copy constructor using specified allocator.
Running time: Linear in the size of `x`

```
map(map<Key,T,Compare,Allocator>&&, const Allocator&);
```

Move constructor using specified allocator.
Running time: Constant when the allocators are the same, linear otherwise

```
map(initializer_list<value_type>, const Compare& = Compare(),
    const Allocator& = Allocator());
```

Initializer list constructor.
Running time: Linear in the size of the initializer list

```
~map();
```

Destructor.
Running time: Linear (destructor is called on every element in the `map`)
Invalidates Iterators and References? Yes

```
map<Key,T,Compare,Allocator>& operator=(const map<Key,T,Compare,Allocator>& x);
```

Copy assignment operator.
Running time: Linear in *S* plus the size of `x`
Invalidates Iterators and References? Yes

```
map<Key,T,Compare,Allocator>& operator=(map<Key,T,Compare,Allocator>&& x);
```

Move assignment operator.
Running time: Constant when the allocators are the same, linear otherwise
Invalidates Iterators and References? Yes

```
map<Key,T,Compare,Allocator>& operator=(initializer_list<value_type> il);
```

Assignment operator accepting an initializer list as right-hand side.
Running time: Linear in *S* plus the number of elements in `il`
Invalidates Iterators and References? Yes

```
void swap(map<Key,T,Compare,Allocator>&);
```

Swaps the contents of the two `map`s.
Running time: Usually constant (just swaps internal pointers), but not required by the standard
Invalidates Iterators and References? 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`.

```
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
```

Not supported for `multimap`. Creates an element in-place in the `map`. Returns a `pair` of an `iterator` referring to the inserted element and a `bool` specifying whether the insertion actually took place.
Running time: Logarithmic
Invalidates Iterators and References? No

```
template <class... Args> iterator emplace(Args&&... args);
```

Only for `multimap`. Creates an element in-place in the `multimap`. Returns an `iterator` referring to the inserted element.
Running time: Logarithmic
Invalidates Iterators and References? No

```
template <class... Args> iterator emplace_hint(const_iterator position,
                                               Args&&... args);
```

Creates an element in-place at `position`, which is just a hint that can be ignored by the implementation.
Running time: Logarithmic
Invalidates Iterators and References? No

```
pair<iterator, bool> insert(const value_type& x);
template <class P> pair<iterator, bool> insert(P&& x);
```

Not supported for `multimap`. Inserts the key/value `pair x` (by copying or moving) if and only if the `map` does not already contain an element with that key. Returns a `pair` of a `iterator` referring to the element with the key of `x` and a `bool` specifying whether the insertion actually took place.
Running time: Logarithmic
Invalidates Iterators and References? No

```
iterator insert(const value_type& x);
template <class P> iterator insert(P&& x);
```

`multimap` only. Inserts the key/value `pair x`. Returns an `iterator` referring to the element with the key of `x`.
Running time: Logarithmic
Invalidates Iterators and References? No

```
iterator insert(const_iterator position, const value_type& x);
template <class P> iterator insert(const_iterator position, P&& x);
```

Inserts the key/value `pair x`. For `multimaps`, always inserts it. For `maps`, inserts it 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`. The `position` parameter is only a hint to the `map`.
Running time: Usually logarithmic, but amortized; constant if `position` is correct
Invalidates Iterators and References? No

```
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
```

Inserts elements from `first` to `last`. For `multimaps` inserts all elements. For `maps` inserts only those for which there is not already a key/value `pair` with that key.
Running time: Usually $N \log (S + N)$, where N is the number of elements inserted, but linear if the inserted range is already sorted correctly
Invalidates Iterators and References? No

```
void insert(initializer_list<value_type> il);
```

Inserts all elements from the initializer list.
Running time: Usually $N \log (S + N)$, where N is the number of elements inserted, but linear if the inserted range is already sorted correctly
Invalidates Iterators and References? No

```
iterator erase(const_iterator position);
```

Removes the element at `position`. Returns an iterator referring to the element following the one that was erased.
Running time: Amortized constant
Invalidates Iterators and References? Invalidates 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.
Running time: Logarithmic
Invalidates Iterators and References? Invalidates only iterators and references to the erased elements

```
iterator erase(const_iterator first, const_iterator last);
```

Removes the elements from `first` to `last`. Returns an iterator referring to the element following the ones that were erased.
Running time: log(*S* + *N*), where *N* is the number of elements erased
Invalidates Iterators and References? Invalidates only iterators and references to the erased elements

```
void clear() noexcept;
```

Erases all elements in the `map`.
Running time: Linear
Invalidates Iterators and References? 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`.

```
T& operator[](const key_type& x);
T& operator[](key_type&& x);
T& at(const key_type& x);
const T& at(const key_type& x) const;
```

`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.
Running time: Logarithmic
Invalidates Iterators and References? No

```
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
```

Returns an `iterator` referring to an element with a key matching `x`. If no element has a key `x`, returns `end()`. Note that for `multimap`s the returned `iterator` can refer to any one of the elements with the specified key.
Running time: Logarithmic
Invalidates Iterators and References? No

```
size_type count(const key_type& x) const;
```

Returns the number of elements with a key matching `x`.
Running time: Logarithmic
Invalidates Iterators and References? No

```
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`.
Running time: Logarithmic
Invalidates Iterators and References? 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`.
Running time: Logarithmic
Invalidates Iterators and References? 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 `iterator`s referring to the first and one-past-the-last elements with keys matching `x`. If the two `iterator`s are equal, there are no elements with key `x`.
Running time: Logarithmic
Invalidates Iterators and References? No

Retrieving Size and Comparison Objects

```
size_type size() const noexcept;
```

The number of elements in the `map`.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
size_type max_size() const noexcept;
```

The maximum number of elements the `map` could hold. Not usually a very useful method, as the number is likely to be quite large.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
bool empty() const noexcept;
```

Returns `true` if the `map` currently has no elements; `false` otherwise.
Running time: Constant
Invalidates Iterators and References? No

```
key_compare key_comp() const;
```

Returns the object used to compare the keys.
Running time: Constant
Invalidates Iterators and References? 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.
Running time: Constant
Invalidates Iterators and References? 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 <class Key, class Compare = less<Key>,
          class 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

```
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.
Running time: Constant

```
explicit set(const Allocator&);
```

Constructor; constructs a `set` of size `0`, with the specified allocator.
Running time: Constant

```
template <class 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.

Running time: Linear logarithmic in the number of elements inserted; linear if the inserted element range is already sorted according to `comp`

```
set(const set<Key,Compare,Allocator>& x);
```

Copy constructor.
Running time: Linear in the size of `x`

```
set(set<Key,Compare,Allocator>&& x);
```

Move constructor.
Running time: Constant when the allocators are the same, linear otherwise

```
set(const set<Key,Compare,Allocator>& x, const Allocator&);
```

Copy constructor using specified allocator.
Running time: Linear in the size of `x`

```
set(set<Key,Compare,Allocator>&&, const Allocator&);
```

Move constructor using specified allocator.
Running time: Constant when the allocators are the same, linear otherwise

```
set(initializer_list<value_type>, const Compare& = Compare(),
    const Allocator& = Allocator());
```

Initializer list constructor.
Running time: Linear in the size of the initializer list

```
~set();
```

Destructor.
Running time: Linear (destructor is called on every element in the `set`)
Invalidates Iterators and References? Yes

```
set<Key,Compare,Allocator>& operator= (const set<Key,Compare,Allocator>& x);
```

Copy assignment operator.
Running time: Linear in *S* plus the size of `x`
Invalidates Iterators and References? Yes

```
set<Key,Compare,Allocator>& operator= (set<Key,Compare,Allocator>&& x);
```

Move assignment operator.
Running time: Constant when the allocators are the same, linear otherwise
Invalidates Iterators and References? Yes

```
set<Key,Compare,Allocator>& operator=(initializer_list<value_type> il);
```

Assignment operator accepting an initializer list as right-hand side.
Running time: Linear in *S* plus the number of elements in `il`
Invalidates Iterators and References? Yes

```
void swap(set<Key,Compare,Allocator>&);
```

Swaps the contents of the two `set`s.
Running time: Usually constant (just swaps internal pointers), but not required by the standard
Invalidates Iterators and References? 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.

```
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
```

Not supported for `multiset`. Creates an element in-place in the `set`. Returns a `pair` of an `iterator` referring to the inserted element and a `bool` specifying whether the insertion actually took place.
Running time: Logarithmic
Invalidates Iterators and References? No

```
template <class... Args> iterator emplace(Args&&... args);
```

Only for `multiset`. Creates an element in-place in the `multiset`. Returns an `iterator` referring to the inserted element.
Running time: Logarithmic
Invalidates Iterators and References? No

```
template <class... Args> iterator emplace_hint(const_iterator position,
                                               Args&&... args);
```

Creates an element in-place at `position`, which is just a hint that can be ignored by the implementation.
Running time: Logarithmic
Invalidates Iterators and References? No

```
pair<iterator,bool> insert(const value_type& x);
pair<iterator,bool> insert(value_type&& x);
```

Not supported for `multiset`. Inserts the element `x` (by copying or moving) 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.
Running time: Logarithmic
Invalidates Iterators and References? No

```
iterator insert(const value_type& x);
iterator insert(value_type&& x);
```

`multiset` only. Inserts the element `x`. Returns an `iterator` referring to the element `x`.
Running time: Logarithmic
Invalidates Iterators and References? No

```
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
```

Inserts the element `x`. For `multiset`s, always inserts it. For `set`s, inserts it if and only if the `set` does not already contain that element. Returns an `iterator` referring to the element. The `position` parameter is only a hint to the `set`.
Running time: Usually logarithmic, but amortized; constant if `position` is correct.
Invalidates Iterators and References? No

```
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
```

Inserts elements from `first` to `last`. For `multiset`s inserts all elements. For `set` inserts only those for which there is not already an element equal to the element to be inserted.
Running time: Usually $N \log (S + N)$, where N is the number of elements inserted, but linear if the inserted range is already sorted correctly
Invalidates Iterators and References? No

```
void insert(initializer_list<value_type> il);
```

Inserts all elements from the initializer list.
Running time: Usually $N \log (S + N)$, where N is the number of elements inserted, but linear if the inserted range is already sorted correctly
Invalidates Iterators and References? No

```
iterator erase(const_iterator position);
```

Removes the element at `position`. Returns an iterator referring to the element following the one that was erased.
Running time: Amortized constant
Invalidates Iterators and References? Invalidates 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.
Running time: Logarithmic
Invalidates Iterators and References? Invalidates only iterators and references to the erased elements

```
iterator erase(const_iterator first, const_iterator last);
```

Removes the elements from `first` to `last`. Returns an iterator referring to the element following the ones that were erased.
Running time: $\log(S + N)$, where N is the number of elements erased
Invalidates Iterators and References? Invalidates only iterators and references to the erased elements

```
void clear() noexcept;
```

Erases all elements in the `set`.
Running time: Linear
Invalidates Iterators and References? Yes

Accessing Elements

```
iterator find(const key_type& x);
const_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 `multiset`s the returned `iterator` can refer to any one of the elements matching `x`.
Running time: Logarithmic
Invalidates Iterators and References? No

```
size_type count(const key_type& x) const;
```

Returns the number of elements matching `x`.
Running time: Logarithmic
Invalidates Iterators and References? No

```
iterator lower_bound(const key_type& x);
const_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`.
Running time: Logarithmic
Invalidates Iterators and References? 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 greater than `x`. Can return `end()` if all elements are less than or equal to `x`.
Running time: Logarithmic
Invalidates Iterators and References? 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 `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`.
Running time: Logarithmic
Invalidates Iterators and References? No

Retrieving Size and Comparison Objects

```
size_type size() const noexcept;
```

The number of elements in the `set`.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
size_type max_size() const noexcept;
```

The maximum number of elements the `set` could hold. Not usually a very useful method, as the number is likely to be quite large.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
bool empty() const noexcept;
```

Returns `true` if the `set` currently has no elements; `false` otherwise.
Running time: Constant
Invalidates Iterators and References? No

```
key_compare key_comp() const;
value_compare value_comp() const;
```

Returns the object used to compare elements in the `set`.
Running time: Constant
Invalidates Iterators and References? No

Unordered Associative Containers/Hash Tables

The unordered associative containers, also called hash tables, include `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`.

Unordered Associative Container typedefs

Unordered associative containers add the following `typedef`s to the common container `typedef`s.

TYPE NAME	DESCRIPTION
`key_type`	The key type with which the container is instantiated.
`mapped_type`	(`unordered_map`s and `unordered_multimap`s only) The "value" type with which the container is instantiated.
`hasher`	The hash function used to calculate the hash of elements.
`key_equal`	A comparator for keys.
`local_iterator`	Provides iteration over the elements in a single bucket.
`const_local_iterator`	Provides `const` iteration over the elements in a single bucket.

unordered_map and unordered_multimap

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

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

Iterators

The `unordered_map` and `unordered_multimap` provide forward iteration.

Template Definition

```
template <class Key, class T, class Hash = hash<Key>,
          class Pred = std::equal_to<Key>,
          class Allocator = std::allocator<std::pair<const Key, T> > >
          class unordered_map;
```

`Key` is the key type and `T` is the value type for elements to be stored in the `unordered_map`, `Hash` is the hash function to calculate the hash of elements, `Pred` is a comparator for keys, and `Allocator` is the type of allocator to be used by the `unordered_map`.

Constructors, Destructors, and Assignment Methods

```
explicit unordered_map(size_type n = see below, const hasher& hf = hasher(),
        const key_equal& eql = key_equal(),
        const allocator_type& a = allocator_type());
```

Default constructor; constructs an `unordered_map` with at least `n` buckets, optionally with the specified hasher, key comparison object, and allocator. The default value for `n` is compiler dependent.
Running time: Constant

```
explicit unordered_map(const Allocator&);
```

Constructor; constructs an `unordered_map` with a default number of buckets, with the specified allocator.
Running time: Constant

```
template <class InputIterator>
    unordered_map(InputIterator f, InputIterator l, size_type n = see below,
        const hasher& hf = hasher(), const key_equal& eql = key_equal(),
        const allocator_type& a = allocator_type());
```

Constructs an `unordered_map` with at least `n` buckets, and inserts the elements from `f` to `l`. It's a method template in order to work on any iterator. The default value for `n` is compiler dependent.
Running time: Average case linear, worst case quadratic

```
unordered_map(const unordered_map& x);
```

Copy constructor.
Running time: Linear in the size of `x`

```
unordered_map(unordered_map&& x);
```

Move constructor.
Running time: Constant when the allocators are the same, linear otherwise

```
unordered_map(const unordered_map& x, const Allocator&);
```

Copy constructor using specified allocator.
Running time: Linear in the size of `x`

```
unordered_map(unordered_map&&, const Allocator&);
```

Move constructor using specified allocator.
Running time: Constant when the allocators are the same, linear otherwise

```
unordered_map(initializer_list<value_type>, size_type n = see below,
        const hasher& hf = hasher(), const key_equal& eql = key_equal(),
        const allocator_type& a = allocator_type());
```

Initializer list constructor. The default value for `n` is compiler dependent.
Running time: Linear in the size of the initializer list

```
~unordered_map();
```

Destructor.
Running time: Linear (destructor is called on every element in the `unordered_map`)
Invalidates Iterators and References? Yes

```
unordered_map& operator=(const unordered_map& x);
```

Copy assignment operator.
Running time: Linear in *S* plus the size of `x`
Invalidates Iterators and References? Yes

```
unordered_map& operator=(unordered_map&& x);
```

Move assignment operator.
Running time: Constant when the allocators are the same, linear otherwise
Invalidates Iterators and References? Yes

```
unordered_map& operator=(initializer_list<value_type> il);
```

Assignment operator accepting an initializer list as right-hand side.
Running time: Linear in *S* plus the number of elements in `il`
Invalidates Iterators and References? Yes

```
void swap(unordered_map&);
```

Swaps the contents of the two `unordered_map`s.
Running time: Usually constant (just swaps internal pointers), but not required by the standard
Invalidates Iterators and References? 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`.

```
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
```

Creates an element in-place in the `unordered_map`. Returns a `pair` of an `iterator` referring to the inserted element and a `bool` specifying whether the insertion actually took place.
Running time: Average case *O*(1); worst case *O*(*S*).
Invalidates Iterators and References? No

```
template <class... Args> iterator emplace_hint(const_iterator position,
                                               Args&&... args);
```

Creates an element in-place at `position`, which is just a hint that can be ignored by the implementation.
Running time: Average case *O*(1); worst case *O*(*S*).
Invalidates Iterators and References? No

```
pair<iterator, bool> insert(const value_type& x);
template <class P> pair<iterator, bool> insert(P&& x);
```

Inserts `x` (by copying or moving). For the `unordered_map`, only inserts it if and only if the `unordered_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.
Running time: Average case *O*(1); worst case *O*(*S*).
Invalidates Iterators and References? No

```
iterator insert(const_iterator hint, const value_type& x);
template <class P> iterator insert(const_iterator hint, P&& x);
```

Inserts `x`. For `unordered_multimap`s, always inserts it. For `unordered_map`s, inserts it if and only if the `unordered_map` does not already contain an element with that key. Returns an `iterator` referring to the element with the key of `x`. The `hint` parameter is only a hint to the `unordered_map`, which can be ignored.
Running time: Average case *O*(1); worst case *O*(*S*).
Invalidates Iterators and References? No

```
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
```

Inserts elements from `first` to `last`. For `unordered_multimap`s inserts all elements. For `unordered_map`s inserts only those for which there is not already a key/value `pair` with that key.
Running time: Average case $O(N)$; worst case $O(N*S+N)$, where N is the number of elements in the range `first` to `last`.
Invalidates Iterators and References? No

```
void insert(initializer_list<value_type> il);
```

Inserts all elements from the initializer list.
Running time: Average case $O(N)$; worst case $O(N*S+N)$, where N is the number of elements in the initializer list.
Invalidates Iterators and References? No

```
iterator erase(const_iterator position);
```

Removes the element at `position`. Returns an iterator referring to the element following the one that was erased.
Running time: Amortized constant
Invalidates Iterators and References? Invalidates only iterators and references to the erased element

```
size_type erase(const key_type& k);
```

Removes all elements in the container with key `k` and returns the number of elements removed.
Running time: Average case $O(1)$; worst case $O(S)$.
Invalidates Iterators and References? Invalidates only iterators and references to the erased elements

```
iterator erase(const_iterator first, const_iterator last);
```

Removes the elements from `first` to `last`. Returns an iterator referring to the element following the ones that were erased.
Running time: On average linear in N, where N is the number of elements erased; worst case $O(S)$.
Invalidates Iterators and References? Invalidates only iterators and references to the erased elements

```
void clear() noexcept;
```

Erases all elements in the `unordered_map`.
Running time: Linear
Invalidates Iterators and References? Yes

Accessing Elements

Most of the methods in this category have `const` and non-`const` versions. If called on a `const unordered_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`.

```
mapped_type& operator[](const key_type& k);
mapped_type& operator[](key_type&& k);
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;
```

`unordered_map`s, but not `unordered_multimap`s, provide standard array access syntax. If no element exists with the specified key, a new element is inserted with that key.
Running time: Average case $O(1)$; worst case $O(S)$.
Invalidates Iterators and References? No

```
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
```

Returns an `iterator` referring to an element with key matching `k`. If no element has key `k`, returns `end()`. Note that for `unordered_multimap`s the returned `iterator` can refer to any one of the elements with the specified key.
Running time: Average case $O(1)$; worst case $O(S)$.
Invalidates Iterators and References? No

```
size_type count(const key_type& k) const;
```

Returns the number of elements with key matching `k`.
Running time: Average case $O(1)$; worst case $O(S)$.
Invalidates Iterators and References? No

```
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
```

Returns a `pair` of `iterator`s referring to the first and one-past-the-last elements with keys matching `k`. If the two `iterator`s are equal, there are no elements with key `k`.
Running time: Worst case $O(S)$.
Invalidates Iterators and References? No

Retrieving Size and Comparison Objects

```
size_type size() const noexcept;
```

The number of elements in the `unordered_map`.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
size_type max_size() const noexcept;
```

The maximum number of elements the `unordered_map` could hold. Not usually a very useful method, as the number is likely to be quite large.

Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
bool empty() const noexcept;
```

Returns `true` if the `unordered_map` currently has no elements; `false` otherwise.
Running time: Constant
Invalidates Iterators and References? No

```
hasher hash_function() const;
```

Returns the hasher object.
Running time: Constant
Invalidates Iterators and References? No

```
key_equal key_eq() const;
```

Returns the object used to compare keys.
Running time: Constant
Invalidates Iterators and References? No

Bucket Interface

```
size_type bucket_count() const noexcept;
```

Returns the number of buckets in the hash table.
Running time: Constant

```
size_type max_bucket_count() const noexcept;
```

Returns the maximum number of buckets that the hash table could contain.
Running time: Constant

```
size_type bucket_size(size_type n) const;
```

Returns the number of elements in the *N*-th bucket.
Running time: Linear in the number of elements in that bucket

```
size_type bucket(const key_type& k) const;
```

Returns the bucket index that contains elements with the given key.
Running time: Constant

```
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;
```

Returns iterators into the bucket with index `n`. Can be used to iterate over the elements in the bucket with index `n`.
Running time: Constant

Hash Policy

```
float load_factor() const noexcept;
```

Returns the average number of elements per bucket.
Running time: Constant

```
float max_load_factor() const noexcept;
```

The hash table automatically attempts to keep the `load_factor()` below the `max_load_factor()`.
Running time: Constant

```
void max_load_factor(float z);
```

Changes the maximum load factor to `z`.
Running time: Constant

```
void rehash(size_type n);
```

Rehashes the container with at least `n` buckets.
Running time: Average linear; worst case quadratic

```
void reserve(size_type n);
```

Same as `rehash(ceil(n/max_load_factor()))`.
Running time: Average linear; worst case quadratic

unordered_set and unordered_multiset

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

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

Iterators

The `unordered_set` and `unordered_multiset` provide forward iteration.

Template Definition

```
template <class Key, class Hash = hash<Key>,
          class Pred = std::equal_to<Key>,
          class Allocator = std::allocator<Key> > class unordered_set;
```

`Key` is the type of the elements to be stored in the `unordered_set`; `Hash` is the hash function to calculate the hash of elements; `Pred` is a comparator for keys; and `Allocator` is the type of allocator to be used by the `unordered_set`.

Constructors, Destructors, and Assignment Methods

```
explicit unordered_set(size_type n = see below,
        const hasher& hf = hasher(), const key_equal& eql = key_equal(),
        const allocator_type& a = allocator_type());
```

Default constructor; constructs an `unordered_set` with at least `n` buckets, optionally with the specified hasher, key comparison object, and allocator. The default value for `n` is compiler dependent.
Running time: Constant

```
explicit unordered_set(const Allocator&);
```

Constructor; constructs an `unordered_set` with a default number of buckets, with the specified allocator.
Running time: Constant

```
template <class InputIterator>
    unordered_set(InputIterator f, InputIterator l, size_type n = see below,
        const hasher& hf = hasher(), const key_equal& eql = key_equal(),
        const allocator_type& a = allocator_type());
```

Constructs an `unordered_set` with at least `n` buckets, and inserts the elements from `f` to `l`. It's a method template in order to work on any iterator. The default value for `n` is compiler dependent.
Running time: Average case linear, worst case quadratic

```
unordered_set(const unordered_set& x);
```

Copy constructor.
Running time: Linear in the size of `x`

```
unordered_set(unordered_set&& x);
```

Move constructor.
Running time: Constant when the allocators are the same, linear otherwise

```
unordered_set(const unordered_set& x, const Allocator&);
```

Copy constructor using specified allocator.
Running time: Linear in the size of `x`

```
unordered_set(unordered_set&& x, const Allocator&);
```

Move constructor using specified allocator.
Running time: Constant when the allocators are the same, linear otherwise

```
unordered_set(initializer_list<value_type>, size_type = see below,
        const hasher& hf = hasher(), const key_equal& eql = key_equal(),
        const allocator_type& a = allocator_type());
```

Initializer list constructor. The default value for `n` is compiler dependent.
Running time: Linear in the size of the initializer list

```
~unordered_set();
```

Destructor.
Running time: Linear (destructor is called on every element in the `unordered_set`)
Invalidates Iterators and References? Yes

```
unordered_set& operator=(const unordered_set& x);
```

Copy assignment operator.
Running time: Linear in *S* plus the size of `x`
Invalidates Iterators and References? Yes

```
unordered_set& operator=(unordered_set&& x);
```

Move assignment operator.
Running time: Constant when the allocators are the same, linear otherwise
Invalidates Iterators and References? Yes

```
unordered_set& operator=(initializer_list<value_type> il);
```

Assignment operator accepting an initializer list as right-hand side.
Running time: Linear in *S* plus the number of elements in `il`
Invalidates Iterators and References? Yes

```
void swap(unordered_set&);
```

Swaps the contents of the two `unordered_set`s.
Running time: Usually constant (just swaps internal pointers), but not required by the standard
Invalidates Iterators and References? No (but the iterators and references now refer to elements in a different container)

Adding and Deleting Elements

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

```
template <class... Args> pair<iterator, bool> emplace(Args&&... args);
```

Creates an element in-place in the `unordered_set`. Returns a `pair` of a `iterator` referring to the inserted element and a `bool` specifying whether the insertion actually took place.
Running time: Average case $O(1)$; worst case $O(S)$.
Invalidates Iterators and References? No

```
template <class... Args> iterator emplace_hint(const_iterator position,
                                               Args&&... args);
```

Creates an element in-place at `position`, which is just a hint that can be ignored by the implementation.
Running time: Average case $O(1)$; worst case $O(S)$.
Invalidates Iterators and References? No

```
pair<iterator, bool> insert(const value_type& x);
pair<iterator, bool> insert(value_type&& x);
```

Not supported for `unordered_multiset`. Inserts the element `x` (by copying or moving) if and only if the `unordered_set` does not already contain that element. Returns a `pair` of a `iterator` referring to the element and a `bool` specifying whether the insert actually took place.
Running time: Average case $O(1)$; worst case $O(S)$.
Invalidates Iterators and References? No

```
iterator insert(const value_type& obj);
iterator insert(value_type&& obj);
```

`unordered_multiset` only. Inserts the element `x`. Returns an `iterator` referring to the element `x`.
Running time: Average case $O(1)$; worst case $O(S)$.
Invalidates Iterators and References? No

```
iterator insert(const_iterator hint, const value_type& x);
iterator insert(const_iterator hint, value_type&& x);
```

Inserts the element `x`. For `unordered_multiset`s, always inserts it. For `unordered_set`s, inserts it if and only if the `unordered_set` does not already contain that element. Returns an `iterator` referring to the element. The `hint` parameter is only a hint to the `set`.
Running time: Average case $O(1)$; worst case $O(S)$.
Invalidates Iterators and References? No

```
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
```

Inserts elements from `first` to `last`. For `unordered_multiset`s inserts all elements. For `unordered_set` inserts only those for which there is not already an element equal to the element to be inserted.
Running time: Average case $O(N)$; worst case $O(N*S+N)$, where N is the number of elements in the range `first` to `last`.
Invalidates Iterators and References? No

```
void insert(initializer_list<value_type> il);
```

Inserts all elements from the initializer list.
Running time: Average case $O(N)$; worst case $O(N*S+N)$, where N is the number of elements in the initializer list.
Invalidates Iterators and References? No

```
iterator erase(const_iterator position);
```

Removes the element at `position`. Returns an iterator referring to the element following the one that was erased.
Running time: Amortized constant
Invalidates Iterators and References? Invalidates only iterators and references to the erased element

```
size_type erase(const key_type& k);
```

Removes all elements in the container matching `k` and returns the number of elements removed.
Running time: Average case $O(1)$; worst case $O(S)$.
Invalidates Iterators and References? Invalidates only iterators and references to the erased elements

```
iterator erase(const_iterator first, const_iterator last);
```

Removes the elements from `first` to `last`. Returns an iterator referring to the element following the ones that were erased.
Running time: On average linear in N, where N is the number of elements erased; worst case $O(S)$.
Invalidates Iterators and References? Invalidates only iterators and references to the erased elements

```
void clear() noexcept;
```

Erases all elements in the `unordered_set`.
Running time: Linear
Invalidates Iterators and References? Yes

Accessing Elements

```
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
```

Returns an `iterator` referring to an element matching `k`. If no element matches `k`, returns `end()`. Note that for `unordered_multiset`s the returned `iterator` can refer to any one of the elements matching `k`.
Running time: Average case $O(1)$; worst case $O(S)$.
Invalidates Iterators and References? No

```
size_type count(const key_type& k) const;
```

Returns the number of elements matching `k`.
Running time: Average case $O(1)$; worst case $O(S)$.
Invalidates Iterators and References? No

```
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
```

Returns a `pair` of `iterator`s referring to the first and one-past the last elements matching `k`. If the two `iterator`s are equal, there are no elements matching `k`.
Running time: Worst case $O(S)$.
Invalidates Iterators and References? No

Retrieving Size and Comparison Objects

```
size_type size() const noexcept;
```

The number of elements in the `unordered_set`.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
size_type max_size() const noexcept;
```

The maximum number of elements the `unordered_set` could hold. Not usually a very useful method, as the number is likely to be quite large.
Running time: Usually constant, but not required to be by the standard
Invalidates Iterators and References? No

```
bool empty() const noexcept;
```

Returns `true` if the `unordered_set` currently has no elements; `false` otherwise.
Running time: Constant
Invalidates Iterators and References? No

```
hasher hash_function() const;
```

Returns the hasher object.
Running time: Constant
Invalidates Iterators and References? No

```
key_equal key_eq() const;
```

Returns the object used to compare keys.
Running time: Constant
Invalidates Iterators and References? No

Bucket Interface

```
size_type bucket_count() const noexcept;
```

Returns the number of buckets in the hash table.
Running time: Constant

```
size_type max_bucket_count() const noexcept;
```

Returns the maximum number of buckets that the hash table could contain.
Running time: Constant

```
size_type bucket_size(size_type n) const;
```

Returns the number of elements in the *N*-th bucket.
Running time: Linear in the number of elements in that bucket.

```
size_type bucket(const key_type& k) const;
```

Returns the bucket index that contains elements with the given key.
Running time: Constant

```
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;
```

Returns iterators into the bucket with index `n`. Can be used to iterate over the elements in the bucket with index `n`.
Running time: Constant

Hash Policy

```
float load_factor() const noexcept;
```

Returns the average number of elements per bucket.
Running time: Constant

```
float max_load_factor() const noexcept;
```

The hash table automatically attempts to keep the `load_factor()` below the `max_load_factor()`.
Running time: Constant

```
void max_load_factor(float z);
```

Changes the maximum load factor to `z`.
Running time: Constant

```
void rehash(size_type n);
```

Rehashes the container with at least `n` buckets.
Running time: Average linear; worst case quadratic

```
void reserve(size_type n);
```

Same as `rehash(ceil(n/max_load_factor()))`.
Running time: Average linear; worst case quadratic

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

```
constexpr bitset() noexcept;
```

Default constructor; constructs a `bitset` of size `N` (as specified in the template parameter) and initializes all bits to zero.
Exceptions: none

```
constexpr bitset(unsigned long long val) noexcept;
```

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.
Exceptions: none

```
template<class charT, class traits, class Allocator>
    explicit bitset(const string& str, string::size_type pos = 0,
                    string::size_type len = string::npos,
                    charT zero = charT('0'), charT one = charT('1'));
```

(This prototype has been simplified for clarity.) 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 `zero` or `one`. If `len < N`, extra high-order bits initialized to 0.
Exceptions: `out_of_range` if `pos > str.size()`
`invalid_argument` if any character in `str` is not zero or `one`

```
template <class charT>
    explicit bitset(const charT* str, string::size_type len = string::npos,
                    charT zero = charT('0'), charT one = charT('1'));
```

(This prototype has been simplified for clarity.) Constructs a `bitset` of size `N` and initializes it from the C-style string `str`, starting at position 0, for `len` characters. The characters in `str` must all be either `zero` or `one`. If `len < N`, extra high-order bits initialized to 0.
Exceptions: `invalid_argument` if any character in `str` is not zero or `one`

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.

```
bitset<N>& set() noexcept;
bitset<N>& set(size_t pos, bool val = true);
```

Sets all the bits in the set to `true`, or sets the specified bit to the specified value.
Exceptions: `out_of_range` if `pos` is not a valid position in the `bitset`

```
bitset<N>& reset() noexcept;
bitset<N>& reset(size_t pos);
```

Sets all the bits, or the specified bit, to `false`.
Exceptions: `out_of_range` if `pos` is not a valid position in the `bitset`

```
bitset<N>& flip() noexcept;
bitset<N>& flip(size_t pos);
```

Toggles all the bits or the specified bit.
Exceptions: `out_of_range` if `pos` is not a valid position in the `bitset`

```
reference operator[](size_t pos);
```

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 complement the bit with `~`.
Exceptions: none

Overloaded Bitwise Operators

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

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

Global functions providing bitwise AND, OR, and exclusive OR. The operands are not modified, and the functions return a new `bitset` containing the result.
Exceptions: none

```
bitset<N> operator<<(size_t pos) const noexcept;
bitset<N> operator>>(size_t pos) const noexcept;
```

Methods providing bitwise left-shift and right-shift operations. Fills in exposed bits with `false`. Doesn't modify `bitset` on which it's called.
Exceptions: none

```
bitset<N> operator~() const noexcept;
```

Method providing bitwise NOT. Doesn't modify `bitset` on which it's called.
Exceptions: none

```
bitset<N>& operator&=(const bitset<N>& rhs) noexcept;
bitset<N>& operator|=(const bitset<N>& rhs) noexcept;
bitset<N>& operator^=(const bitset<N>& rhs) noexcept;
bitset<N>& operator<<=(size_t pos) noexcept;
bitset<N>& operator>>=(size_t pos) noexcept;
```

Methods providing bitwise assignment operators. Modify the `bitset` on which they are called.
Exceptions: none

Stream Operators

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

```
ostream& operator<<(ostream& os, const bitset<N>& x);
istream& operator>>(istream& is, bitset<N>& x);
```

Global insertion and extraction operators for `bitset`s. Reads and writes `bitset`s as strings of 0 or 1.
Exceptions: None, unless the streams throw

Obtaining Information about Bits in the Bitset

```
constexpr bool operator[](size_t pos) const;
bool test(size_t pos) const;
```

Returns the value of the bit at `pos`.
Exceptions: `out_of_range` if `pos` is not a valid position in the `bitset`

```
constexpr size_t size() noexcept;
```

Returns the number of bits in the `bitset`.
Exceptions: None

```
size_t count() const noexcept;
```

Returns the number of bits in the `bitset` set to `true`.
Exceptions: None

```
bool all() const noexcept;
```

Returns `true` if all bits in the `bitset` are set to `true`.
Exceptions: None

```
bool any() const noexcept;
```

Returns `true` if one or more bits in the `bitset` are set to `true`.
Exceptions: None

```
bool none() const noexcept;
```

Returns `true` if all bits in the `bitset` are set to `false`.
Exceptions: None

```
bool operator==(const bitset<N>& rhs) const noexcept;
bool operator!=(const bitset<N>& rhs) const noexcept;
```

Bit-by-bit comparison between two `bitsets`.
Exceptions: None

```
unsigned long to_ulong() const;
unsigned long long to_ullong() const;
string to_string(charT zero = charT('0'), charT one = charT('1')) const;
```

Converts the `bitset` to an `unsigned long` or to a `string`.
Exceptions: `overflow_error` if the `bitset` value cannot be represented as `unsigned long` or `unsigned long long`

string

The `string` and `wstring` classes are `typedefs` 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.

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

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

Default constructor; constructs a `string` of size `0`, optionally with the specified allocator.
Exceptions: None (unless the allocator throws an exception)

```
string(const charT* s, size_type n, const Allocator& a = Allocator());
string(const charT* s, const Allocator& 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.
Exceptions: None (unless the allocator throws an exception)

```
string(size_type n, charT c, const Allocator& a = Allocator());
```

Constructs a `string` containing `n` copies of character `c`.
Exceptions: None (unless the allocator throws an exception)

```
template<class InputIterator>
    string(InputIterator first, InputIterator last,
           const Allocator& a = Allocator());
```

Constructs a `string` and inserts the elements from `first` to `last`. It's a method template in order to work on any iterator.
Exceptions: 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`.
Exceptions: `out_of_range` if `pos > str.size()`

```
string(string&& str) noexcept;
```

Move constructor.
Exceptions: None (unless the allocator throws an exception)

```
string(const string&, const Allocator&);
```

Copy constructor using specified allocator.
Exceptions: None (unless the allocator throws an exception)

```
string(string&&, const Allocator&);
```

Move constructor using specified allocator.
Exceptions: None (unless the allocator throws an exception)

```
string(initializer_list<charT>, const Allocator& = Allocator());
```

Initializer list constructor.
Exceptions: None (unless the allocator throws an exception)

```
~string();
```

Destructor.
Invalidates Iterators and References? Yes
Exceptions: none

```
string& operator=(const string& str);
string& operator=(const charT* s);
string& operator=(charT c);
```

Copy assignment operator: replaces the contents of the target `string` object with a copy of `str`, `s`, or `c`.
Invalidates Iterators and References? Yes
Exceptions: none

```
string& operator=(string&& str) noexcept;
```

Move assignment operator.
Invalidates Iterators and References? Yes
Exceptions: none

```
string& operator=(initializer_list<charT> il);
```

Assignment operator accepting an initializer list as right-hand side.
Invalidates Iterators and References? Yes
Exceptions: none

```
template<class InputIterator>
    string& assign(InputIterator first, InputIterator last);
```

Removes all the current elements and inserts all the elements from `first` to `last`.
Invalidates Iterators and References? Yes
Exceptions: none

```
string& assign(const string& str);
string& assign(string&& str) noexcept;
string& assign(const string& str, size_type pos, size_type n);
string& assign(const charT* s, size_type n);
string& assign(const charT* s);
string& assign(size_type n, charT 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`.
Invalidates Iterators and References? Yes
Exceptions: `out_of_range` if `pos > str.size()`

```
string& assign(initializer_list<charT> il);
```

Removes all the current elements and inserts all elements of the initializer list.
Invalidates Iterators and References? Yes
Exceptions: none

```
void swap(string& str);
```

Swaps the contents of the two `string`s.
Invalidates Iterators and References? Yes
Exceptions: none

Adding and Deleting Characters

```
iterator insert(const_iterator p, charT 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.
Invalidates Iterators and References? Yes
Exceptions: None

```
iterator insert(const_iterator p, size_type n, charT c);
```

Inserts `n` copies of `c` before the character at `p`, shifting all subsequent characters to make room. Returns an `iterator` referring to the character inserted.
Invalidates Iterators and References? Yes
Exceptions: None

```
template<class InputIterator>
    iterator insert(const_iterator p, InputIterator first, InputIterator last);
```

Inserts all characters in the range `[first,last)` before the character at `p`. Returns an `iterator` referring to the character inserted.
Invalidates Iterators and References? Yes
Exceptions: None

```
string& insert(size_type pos, const charT* s);
string& insert(size_type pos, const string& str);
```

Inserts all the characters from `str` or `s` starting at position `pos`.
Invalidates Iterators and References? Yes
Exceptions: `out_of_range` if `pos > size()`

```
string& insert(size_type pos1, const string& str, size_type pos2, size_type n);
string& insert(size_type pos, const charT* 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.
Invalidates Iterators and References? Yes
Exceptions: `out_of_range` if `pos > size()` or `pos2 > size()`

```
string& insert(size_type pos, size_type n, charT c);
```

Inserts `n` copies of `c` starting at `pos` in the target `string`.
Invalidates Iterators and References? Yes
Exceptions: None

```
iterator insert(const_iterator p, initializer_list<charT>);
```

Inserts all characters from the initializer list into the `string` at position `p`. Returns an `iterator` referring to the character inserted.
Invalidates Iterators and References? Yes
Exceptions: None

```
string& erase(size_type pos = 0, size_type n = npos);
```

Removes `n` characters starting at `pos`.
Invalidates Iterators and References? Yes
Exceptions: `out_of_range` if `pos > size()`

```
iterator erase(const_iterator p);
```

Removes the character at `p`. Returns an `iterator` referring to the element following the one that was erased.
Invalidates Iterators and References? Yes
Exceptions: None

```
iterator erase(const_iterator first, const_iterator last);
```

Removes the characters in the range `[first,last)`. Returns an `iterator` referring to the element following the ones that were erased.
Invalidates Iterators and References? Yes
Exceptions: None

```
void pop_back();
```

Removes the last character from the `string`.
Invalidates Iterators and References? Yes
Exceptions: None

```
void clear() noexcept;
```

Removes all characters in the `string`.
Invalidates Iterators and References? Yes
Exceptions: None

```
string& replace(size_type pos, size_type n1, const string& str);
string& replace(size_type pos, size_type n1, const charT* s);
string& replace(const_iterator i1, const_iterator i2, const string& str);
string& replace(const_iterator i1, const_iterator i2, const charT* s);
```

The first two forms remove `n1` characters from the target `string` starting at 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`.
Invalidates Iterators and References? Yes
Exceptions: `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 charT* s, size_type n2);
string& replace(const_iterator i1, const_iterator i2, const charT* 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`.
Invalidates Iterators and References? Yes
Exceptions: `out_of_range` if `pos > size()` or `pos2 > str.size()`

```
string& replace(size_type pos, size_type n1, size_type n2, charT c);
string& replace(const_iterator i1, const_iterator i2, size_type n2, charT 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`.
Invalidates Iterators and References? Yes
Exceptions: `out_of_range` if `pos > size()`

```
template<class InputIterator>
    string& replace(const_iterator i1, const_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`.
Invalidates Iterators and References? Yes
Exceptions: None

```
string& replace(const_iterator i1, const_iterator i2,
                initializer_list<charT>);
```

Removes the characters in the range `[i1,i2)`. Inserts the characters from the initializer list starting at `i1`.
Invalidates Iterators and References? Yes
Exceptions: 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 charT* s);
string& append(const charT* s);
string& append(const charT* s, size_type n);
string& operator+=(charT c);
string& append(size_type n, charT c);
string& operator+=(initializer_list<charT>);
string& append(initializer_list<charT>);
```

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, a character, or an initializer list. The `append()` method 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.
Invalidates Iterators and References? Yes
Exceptions: `out_of_range` if `pos > size()`

```
template<class InputIterator>
    string& append(InputIterator first, InputIterator last);
```

Adds the characters in the range `[first,last)` to the end of the target `string`.
Invalidates Iterators and References? Yes
Exceptions: None

```
void push_back(charT c);
```

Appends character `c` to the target `string`.
Invalidates Iterators and References? Yes
Exceptions: 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.

```
reference operator[](size_type pos);
const_reference operator[](size_type pos) const;
```

Array syntax for character access. Does not perform bounds checking.
Invalidates Iterators and References? No
Exceptions: None

```
reference at(size_type n);
const_reference at(size_type n) const;
```

Method for character access.
Invalidates Iterators and References? No
Exceptions: `out_of_range` if `n >= size()`

```
const charT& front() const;
charT& front();
const charT& back() const;
charT& back();
```

Methods for accessing the first or last character in the `string`.
Invalidates Iterators and References? No
Exceptions: None

Obtaining Characters

```
const charT* c_str() const noexcept;
const charT* data() const noexcept;
```

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`.
Invalidates Iterators and References? No
Exceptions: None

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

Copies `n` characters in the target `string` starting at `pos` into the character buffer pointed to by `s`.
Invalidates Iterators and References? No
Exceptions: `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`.
Invalidates Iterators and References? No
Exceptions: `out_of_range` if `pos > size()`

Retrieving and Setting Size and Capacity

None of the following methods throw an exception.

```
size_type size() const noexcept;
size_type length() const noexcept;
```

The number of characters in the `string`.
Invalidates Iterators and References? No

```
size_type max_size() const noexcept;
```

The maximum number of characters the `string` could hold. Not usually a very useful method, as the number is likely to be quite large.
Invalidates Iterators and References? No

```
void resize(size_type sz, charT c);
void resize(size_type sz);
```

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.
Invalidates Iterators and References? Yes

```
size_type capacity() const noexcept;
```

The number of characters the `string` could hold without a reallocation.
Invalidates Iterators and References? No

```
bool empty() const noexcept;
```

Returns `true` if the `string` currently has no characters; `false` otherwise.
Invalidates Iterators and References? No

```
void reserve(size_type n = 0);
```

Changes the capacity of the string to `n`. Does not change the size of the `string`.
Invalidates Iterators and References? Yes

```
void shrink_to_fit();
```

Non-binding request to reduce `capacity()` to `size()`.
Invalidates Iterators and References? Yes

Searching

None of the following methods throw an exception, and none of the following methods invalidate iterators and references.

```
size_type find (charT s, size_type pos = 0) const noexcept;
size_type rfind(charT s, size_type pos = npos) const noexcept;
```

Returns the index of the first character in the target `string` matching `s`. The search starts at `pos` and moves forward with `find()` or backward with `rfind()`. Returns `npos` if no match is found.

```
size_type find (const string& str, size_type pos = 0) const noexcept;
size_type find (const charT* s, size_type pos, size_type n) const;
size_type find (const charT* s, size_type pos = 0) const;
size_type rfind(const string& str, size_type pos = npos) const noexcept;
size_type rfind(const charT* s, size_type pos, size_type n) const;
size_type rfind(const charT* s, size_type pos = npos) const;
```

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.

```
size_type find_first_of(const string& str, size_type pos = 0) const noexcept;
size_type find_first_of(const charT* s, size_type pos, size_type n) const;
size_type find_first_of(const charT* s, size_type pos = 0) const;
size_type find_first_of(charT s, size_type pos = 0) const noexcept;
size_type find_last_of (const string& str, size_type pos = npos) const noexcept;
size_type find_last_of (const charT* s, size_type pos, size_type n) const;
size_type find_last_of (const charT* s, size_type pos = npos) const;
size_type find_last_of (charT s, size_type pos = npos) const noexcept;
```

Returns the index in the target `string` 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.

```
size_type find_first_not_of(const string& str, size_type pos = 0) const noexcept;
size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
size_type find_first_not_of(charT s, size_type pos = 0) const noexcept;
size_type find_last_not_of (const string& str, size_type pos = npos)
    const noexcept;
size_type find_last_not_of (const charT* s, size_type pos, size_type n) const;
size_type find_last_not_of (const charT* s, size_type pos = npos) const;
size_type find_last_not_of (charT s, size_type pos = npos) const noexcept;
```

Like `find_first_of()` and `find_last_of()`, except finds the first or last character not in `str` or `s`.

Comparisons

In addition to the standard comparison operators such as `operator==` and `operator!=`, `string`s provide a `compare()` method with the following prototypes. None of the following methods invalidate iterators and references.

```
int compare(const string& str) const noexcept;
int compare(const charT* s) const;
```

Performs a character-by-character comparison between `str` or `s` and the target `string`. Returns 0 if the two strings 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`.
Exceptions: None

```
int compare(size_type pos1, size_type n1, const string& str) const;
int compare(size_type pos1, size_type n1, const charT* 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.
Exceptions: `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 charT* 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`.
Exceptions: `out_of_range` if `pos1 > size()` or `pos2 > size()`

Concatenating strings

```
string operator+(const string& lhs, const string& rhs);
string operator+(string&& lhs, const string& rhs);
string operator+(const string& lhs, string&& rhs);
string operator+(string&& lhs, string&& rhs);
string operator+(const charT* lhs, const string& rhs);
string operator+(const charT* lhs, string&& rhs);
string operator+(charT lhs, const string& rhs);
string operator+(charT lhs, string&& rhs);
string operator+(const string& lhs, const charT* rhs);
```

```
string operator+(string&& lhs, const charT* rhs);
string operator+(const string& lhs, charT rhs);
string operator+(string&& lhs, charT 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.
Invalidates Iterators and References? No
Exceptions: None

Streaming strings

```
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.
Invalidates Iterators and References? Yes
Exceptions: None

```
ostream& operator<<(ostream& os, const string& str);
```

Writes the characters in `str` to `os`.
Invalidates Iterators and References? No
Exceptions: None

```
istream& getline(istream& is, string& str);
istream& getline(istream&& is, string& str);
istream& getline(istream& is, string& str, charT delim);
istream& getline(istream&& is, string& str, charT delim);
```

Reads characters from `is` and appends them to `str` until `\n` or `delim` is found. `delim` is not appended to `str`.
Invalidates Iterators and References? Yes
Exceptions: None

Numeric Conversions

```
int stoi(const string& str, size_t *idx = 0, int base = 10);
long stol(const string& str, size_t *idx = 0, int base = 10);
unsigned long stoul(const string& str, size_t *idx = 0, int base = 10);
long long stoll(const string& str, size_t *idx = 0, int base = 10);
unsigned long long stoull(const string& str, size_t *idx = 0, int base = 10);
float stof(const string& str, size_t *idx = 0);
double stod(const string& str, size_t *idx = 0);
long double stold(const string& str, size_t *idx = 0);
```

Converts the `string` to a numerical value.
Exceptions: `invalid_argument` if no conversion could be performed, `out_of_range` if the converted value is outside the range representable by the return type

```
string to_string(int val);
string to_string(unsigned val);
```

```
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);
```

Converts a numerical value to a `string`.
Exceptions: None

ALGORITHMS

All the algorithms are templatized functions on one or more type parameters. For simplicity, this section doesn'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.
`Compare`	A lambda expression, function pointer, or functor that compares two elements, returning `true` if the first is less than the second, `false` otherwise.
`Predicate`	A lambda expression, function pointer, or functor that returns `true` or `false` when passed an element as its single argument.

continues

(continued)

TYPE NAME	MEANING
`BinaryPredciate`	A lambda expression, 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 lambda expression, function pointer, or functor that takes an element and returns an element.
`BinaryOperation`	A lambda expression, function pointer, or functor that takes two elements and returns a single element.
`Function`	A lambda expression, function pointer, or functor that takes one element. The return type is irrelevant.
`RandomNumberGenerator`	A lambda expression, function pointer, or functor that takes one integer argument `n` and returns an integer in the range `[0,n)`.
`Generator`	A lambda expression, 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

```
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);
```

Returns 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.
Running time: Constant

```
T min(initializer_list<T> t);
T min(initializer_list<T> t, Compare comp);
T max(initializer_list<T> t);
T max(initializer_list<T> t, Compare comp);
```

Returns the minimum or maximum of the values in the initializer list, using `operator<` or the supplied comparison callback to compare them.
Returns: The minimum or maximum value.
Running time: Linear

```
pair<const T&, const T&> minmax(const T& a, const T& b);
pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```

Returns a `pair` containing a reference to both the minimum and maximum of two values, using `operator<` or the supplied comparison callback to compare them.
Returns: A `pair` containing a reference to the minimum and maximum value.
Running time: Constant

```
pair<T, T> minmax(initializer_list<T> t);
pair<T, T> minmax(initializer_list<T> t, Compare comp);
```

Returns a `pair` containing both the minimum and maximum of the values in the initializer list, using `operator<` or the supplied comparison callback to compare them.
Returns: A `pair` containing the minimum and maximum value.
Running time: Linear

```
void swap(T& a, T& b);
```

Exchanges two values.
Returns: void
Running time: Constant

Non-Modifying Algorithms

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

Search Algorithms

```
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`.
Returns: An iterator referring to the first matching element, or `last` if no match is found.
Requires Sorted Sequence? No
Running time: Linear

```
InputIterator find_if_not(InputIterator first, InputIterator last,
                          Predicate pred);
```

Finds the first element that does not causes `pred` to return `true`.
Returns: An iterator referring to the first non-matching element, or `last` if no match is found.
Requires Sorted Sequence? No
Running time: Linear

```
InputIterator find_first_of(InputIterator first1, InputIterator last1,
                          ForwardIterator first2, ForwardIterator last2);
InputIterator find_first_of(InputIterator first1, InputIterator last1,
                          ForwardIterator first2, ForwardIterator 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`.
Returns: An iterator referring to the first matching element, or `last1` if no match is found.
Requires Sorted Sequence? No
Running time: 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`.
Returns: An iterator referring to the first element of the matching pair, or `last` if no match is found.
Requires Sorted Sequence? No
Running time: Linear

```
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)`.
Returns: An iterator referring to the first element of the matching subsequence in the range `[first1,last1)`, or `last1` if no matching subsequence is found.
Requires Sorted Sequence? No
Running time: 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 `count` consecutive elements equal to `value` in the range `[first,last)`. Compares elements with `operator==` or `pred`.
Returns: An iterator referring to the first element of the matching subsequence, or `last` if no matching subsequence is found.
Requires Sorted Sequence? No
Running time: Quadratic

```
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 last, const T& value, Compare comp);
```

Finds the beginning (`lower_bound()`) end (`upper_bound()`) or both sides (`equal_range()`) of the range including `value`. Compares elements with `operator<` or `comp`.
Returns: `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 `pair` of iterators that `lower_bound()` and `upper_bound()` would return separately.
Requires Sorted Sequence? Yes
Running time: 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: Returns `true` or `false` specifying whether `value` is in the range `[first,last)`.
Requires Sorted Sequence? Yes
Running time: Logarithmic for random access iterators; Linear otherwise

```
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: Returns an iterator referring to the minimum or maximum element.
Requires Sorted Sequence? No
Running time: Linear

```
pair<ForwardIterator, ForwardIterator> minmax_element(
    ForwardIterator first, ForwardIterator last);
pair<ForwardIterator, ForwardIterator> minmax_element(
    ForwardIterator first, ForwardIterator last, Compare comp);
```

Finds the minimum and maximum element in the range `[first,last)`, comparing elements with `operator<` or `comp`.
Returns: Returns a pair containing iterators referring to the minimum and maximum element.

Requires Sorted Sequence? No
Running time: Linear

```
bool all_of(InputIterator first, InputIterator last, Predicate pred);
```

Returns: Returns `true` if `pred` returns `true` for all elements in the range `[first, last)`.
Requires Sorted Sequence? No
Running time: Linear

```
bool any_of(InputIterator first, InputIterator last, Predicate pred);
```

Returns: Returns `true` if `pred` returns `true` for at least one element in the range `[first, last)`.
Requires Sorted Sequence? No
Running time: Linear

```
bool none_of(InputIterator first, InputIterator last, Predicate pred);
```

Returns: Returns `true` if `pred` returns `false` for all elements in the range `[first, last)`.
Requires Sorted Sequence? No
Running time: Linear

```
ForwardIterator partition_point(ForwardIterator first, ForwardIterator last,
                                Predicate pred);
```

Returns: An iterator such that all elements before this iterator return `true` for a predicate `pred`, and all elements after this iterator return `false` for `pred`.
Requires Sorted Sequence? No
Running time: Logarithmic

Numerical Processing Algorithms

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

```
difference_type count(InputIterator first, InputIterator last, const T& value);
difference_type count_if(InputIterator first, InputIterator last,
                         Predicate pred);
```

Counts the number of elements matching `value` with `operator==`, or for which `pred` returns `true`.
Returns: The number of matching elements.
Running time: 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`.

Returns: The accumulated value.
Running time: Linear

```
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 sequences, accumulating the result with `operator+` or `binary_op1`. 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)`.
Returns: The accumulated value.
Running time: 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 non-modifying algorithm). The sum can be calculated with `operator+` or `binary_op`.
Returns: The past-the-end iterator of the sequence starting at `result`.
Running time: 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 non-modifying algorithm). The difference can be calculated with `operator-` or `binary_op`.
Returns: The past-the-end iterator of the sequence starting at `result`.
Running time: Linear

```
void iota(ForwardIterator first, ForwardIterator last, T value);
```

Writes to each element in the range `[first,last)` the value of `value` and increments `value` (`++value`). Technically, this is a numerical processing algorithm, but it's a modifying one.
Returns: `void`
Running time: Linear

Comparison Algorithms

```
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)`.
Returns: `true` if the two ranges are equal; `false` otherwise.
Running time: 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)`.
Returns: 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`.
Running time: 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 of the 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<` or `comp`.
Returns: `true` if the sequences are lexicographically equal; `false` otherwise.
Running time: Linear

Operational Algorithms

```
Function for_each(InputIterator first, InputIterator last, Function f)
```

Executes `f` on each element in the sequence.
Returns: `f`, which can be used to accumulate information about the elements.
Running time: Linear

Modifying Algorithms

Unlike the non-modifying 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.

```
OutputIterator transform(InputIterator first, InputIterator last,
        OutputIterator result, UnaryOperation op);
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, 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)`.
Returns: An iterator referring to one past the end of the new sequence starting with `result`.
Running time: 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: Returns the past-the-end iterator of the new sequence beginning at `result`.
Running time: 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: Returns the start iterator of the new sequence that ends with `result`.
Running time: Linear

```
OutputIterator copy_n(InputIterator first, Size n, OutputIterator result);
```

Copies `n` elements starting at `first` to the range beginning with `result` may not be in the range `[first,last)`, but the ranges may otherwise overlap.
Returns: Returns the past-the-end iterator of the new sequence beginning at `result`.
Running time: Linear

```
OutputIterator copy_if(InputIterator first, InputIterator last,
                       OutputIterator result, Predicate pred);
```

Copies elements from the range `[first, last)` that cause `pred` to return `true` to the range beginning with `result`.
`result` may not be in the range `[first,last)`, but the ranges may otherwise overlap.
Returns: Returns the past-the-end iterator of the new sequence beginning at `result`.
Running time: Linear

```
pair<OutputIterator1, OutputIterator2> partition_copy(
        InputIterator first, InputIterator last,
        OutputIterator1 out_true, OutputIterator2 out_false, Predicate pred);
```

Copies elements from one sequence to two different sequences. The target sequence is selected based on the result of a predicate `pred`, either `true` or `false`.
Returns: A `pair p` such that `p.first` is the end of the output range beginning at `out_true`, and `p.second` is the end of the output range beginning at `out_false`.
Running time: Linear

```
OutputIterator move(InputIterator first, InputIterator last,
                    OutputIterator result);
```

Same as `copy()`, but uses move semantics.
Returns: Returns the past-the-end iterator of the new sequence beginning at `result`.
Running time: Linear

```
BidirectionalIterator2 move_backward(BidirectionalIterator1 first,
        BidirectionalIterator1 last, BidirectionalIterator2 result);
```

Same as `copy_backward()`, but uses move semantics.
Returns: Returns the start iterator of the new sequence that ends with `result`.
Running time: Linear

```
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2);
```

Swaps two elements referred to by iterators `a` and `b`, or two ranges `[first1,last1)` and the range beginning at `first2`.
Returns: `swap_ranges()` returns the past-the-end iterator of the range beginning at `first2`.
Running time: 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 replace_copy(InputIterator first, InputIterator last,
        OutputIterator result, const T& old_value, const T& new_value);
OutputIterator replace_copy_if(InputIterator first, InputIterator 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 last two forms modify the range beginning with `result`. The ranges cannot overlap.
Returns: The in-place forms return nothing.
The copy forms return the past-the-end iterator of the new range beginning with `result`.
Running time: Linear

```
void fill(ForwardIterator first, ForwardIterator last, const T& value);
OutputIterator 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`.
Returns: `fill_n()` returns `first+n` for non-negative values of `n` and `first` for negative values.
Running time: Linear

```
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
```

Like `fill()` and `fill_n()`, except calls `gen` to generate values.
Returns: `generate_n()` returns `first+n` for non-negative values of `n` and `first` for negative values
Running time: Linear

```
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 last 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 as well.
Returns: The past-the-end iterator of the destination range.
Running time: 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`.
Returns: The past-the-end iterator of the destination range.
Running time: Linear

```
void reverse(BidirectionalIterator first, BidirectionalIterator last);
OutputIterator reverse_copy(BidirectionalIterator 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.

Returns: The second form returns the new past-the-end iterator of the range beginning with `result`.
Running time: Linear

```
ForwardIterator 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.
Returns: The first form returns `first+(last-middle)`. The second form returns `result+(last-first)`.
Running time: Linear

```
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last,
                      Compare comp);
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);
bool prev_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 `lexicographical_compare()`, using `operator<` or `comp`. Successive calls to one or the other will permute the sequence into all possible permutations of elements.
Returns: Returns `true` if there is another "next" or "previous" permutation.
Running time: Linear

```
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                    ForwardIterator2 first2);
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                    ForwardIterator2 first2, BinaryPredicate pred);
```

Used to check if there exists a permutation. This is technically a non-modifying algorithm, but it belongs together with the other permutation algorithms, so it is shown here.
Returns: `true` if there exists a permutation of the elements in the range `[first2,first2 +(last1-first1))`, such that `equal(first1, last1, first2)` returns `true` or `equal(first1, last1, first2, pred)` returns `true`; otherwise, returns `false`.
Running time: $O(N^2)$ worst case, with `n=last1-first1`

Sorting Algorithms

Most sorting algorithms have two forms: The first uses `operator<` and the second takes a comparison callback, `comp`. Both prototypes are shown here, but the `comp` parameter is not explained every time; it always means the same: `comp` is a custom sorting criterion.

```
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

```
void sort(RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

Sorts the range `[first,last)` in-place. The `stable_sort()` algorithm preserves the order of duplicate elements.
Returns: void
Running time: *N* log *N* in general, but quadratic in worst case

```
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
        RandomAccessIterator last);
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
        RandomAccessIterator last, Compare comp);
RandomAccessIterator partial_sort_copy(InputIterator first, InputIterator last,
        RandomAccessIterator result_first, RandomAccessIterator result_last);
RandomAccessIterator partial_sort_copy(InputIterator first, InputIterator last,
        RandomAccessIterator result_first, RandomAccessIterator result_last,
        Compare comp);
```

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)`.
Returns: void or the past-the-end iterator of the new range.
Running time: *N* log *N*

```
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last, Compare comp);
```

Relocates the element referred to by *N*th in the range `[first,last)` as if the entire range were sorted.
Also partitions the range as if `partition()` had been called.
Returns: void
Running time: Linear in general; quadratic in worst case

```
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2, OutputIterator result);
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2, OutputIterator result,
        Compare comp);
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle,
         BidirectionalIterator last);
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle,
         BidirectionalIterator last, Compare comp);
```

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.
Returns: The past-the-end iterator of the merged range.
Running time: Linear

```
bool is_sorted(ForwardIterator first, ForwardIterator last);
bool is_sorted(ForwardIterator first, ForwardIterator last, Compare comp);
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last,
                                Compare comp);
```

Checks whether the range `[first,last)` is sorted.
Returns: `is_sorted()` returns `true` when the range is sorted, `false` otherwise.
`is_sorted_until()` returns an iterator until where the range is sorted.
Running time: Linear

```
void make_heap(RandomAccessIterator first, RandomAccessIterator last);
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
void push_heap(RandomAccessIterator first, RandomAccessIterator last);
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
             Compare comp);
RandomAccessIterator is_heap_until(RandomAccessIterator first,
                                   RandomAccessIterator last);
RandomAccessIterator is_heap_until(RandomAccessIterator first,
                                   RandomAccessIterator last, Compare comp);
```

`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)`.
`sort_heap()` turns the heap in `[first,last)` into a fully sorted sequence.
`is_heap()` returns whether the range `[first,last)` is a heap.
`is_heap_until()` returns until where the range `[first,last)` is a heap.
Returns: `is_heap()` returns `true` if the range is a heap.
`is_heap_until()` returns an iterator until where the range is a heap.
Running time: `make_heap()` is linear, `push_heap()` and `pop_heap()` are logarithmic, `sort_heap()` is *N* log *N*, and `is_heap()` and `is_heap_until()` are linear

```
ForwardIterator partition(ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

```
BidirectionalIterator stable_partition(BidirectionalIterator first,
                          BidirectionalIterator last, Predicate pred);
```

Sorts the range `[first,last)` such that all elements for which `pred` returns `true` are before all elements for which it returns `false`.
`stable_partition()` preserves the original order of the elements within a partition.
Returns: 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.
Running time: Linear

```
bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);
```

Returns whether the range `[first,last)` is partitioned or not. This is technically a non-modifying algorithm, but it belongs together with the other partition algorithms, so it is shown here.
Returns: `true` if all elements for which `pred` returns `true` are before all elements for which `pred` returns `false`.
Running time: Linear

```
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
                    RandomNumberGenerator&& rand);
void shuffle(RandomAccessIterator first, RandomAccessIterator last,
             UniformRandomNumberGenerator&& 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*).
The third form requires a function pointer that returns a random unsigned integer.
Returns: void
Running time: Linear

Set Algorithms

All the `set` algorithms have two forms: The first uses `operator==` and the second takes a comparison callback, `comp`. Both prototypes are shown here, but the `comp` parameter is not explained every time; it always means the same: `comp` is a custom comparison criterion.

```
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2, Compare comp);
```

Determines if the sequence `[first2,last2)` is a subset of `[first1,last1)`.
Returns: `true` if the range `[first1,last1)` contains all elements in the range `[first2,last2)`.
Running time: Linear

```
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result);
```

```
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp);
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                                OutputIterator result);
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                                OutputIterator result, Compare comp);
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator result);
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator result, Compare comp);
OutputIterator set_symmetric_difference(InputIterator1 first1,
        InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
        OutputIterator result);
OutputIterator set_symmetric_difference(InputIterator1 first1,
        InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
        OutputIterator result, Compare comp);
```

Performs the specified set operation on two ranges. The resulting elements are copied to the range starting with `result`.
Returns: The past-the-end iterator of the `result` range.
Running time: 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`.

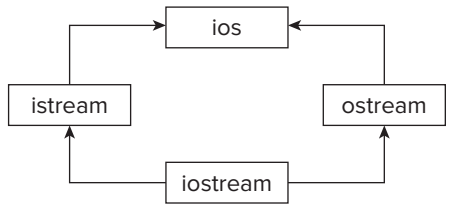

FIGURE 1

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), unbuffered.
`std::wclog`	Standard error (wide characters), buffered.

Predefined stream Manipulators

Manipulators are used to change a run-time behavior of a stream and are described in Chapter 15.

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).

continues

(continued)

MANIPULATOR	DESCRIPTION
`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.
`hexfloat`	A combination of `fixed` and `scientific`.
`defaultfloat`	Remove the `fixed` and `scientific` flag.

ios

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

Stream Status Methods

```
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.

```
explicit operator bool() const;
```

Equivalent to `good()`.

```
bool operator!() const;
```

Equivalent to `fail()`.

```
void clear(iostate state = goodbit);
```

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

```
istream& operator>>(bool& n);
istream& operator>>(short& n);
istream& operator>>(unsigned short& n);
istream& operator>>(int& n);
istream& operator>>(unsigned int& n);
istream& operator>>(long& n);
istream& operator>>(unsigned long& n);
istream& operator>>(long long& n);
istream& operator>>(unsigned long long& n);
istream& operator>>(float& f);
istream& operator>>(double& f);
istream& operator>>(long double& f);
istream& operator>>(void*& p);
```

Parses numerical/Boolean data from the stream.

```
istream& operator>>(istream&, char*);
istream& operator>>(istream&, unsigned char*);
istream& operator>>(istream&, signed char*);
```

Parses character array (C-style string) data from the stream.

```
istream& operator>>(istream&, char&);
istream& operator>>(istream&, unsigned char&);
istream& operator>>(istream&, signed char&);
```

Parses character data from the stream.

```
istream& operator>>(streambuf* sb);
```

Parses characters from the stream and stores them in the stream buffer `sb`.

Unformatted Input

```
int_type get();
```

Extracts a single character if one exists. If no character exists, the result will be `eof`.

```
istream& get(char_type& c);
```

Extracts a single character and returns a reference to the stream.

```
istream& get(char_type* s, streamsize n, char_type 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_type* s, streamsize n);
```

Extracts up to `n-1` characters into the character array pointed to by `s` until end-of-file is reached.

```
istream& get(streambuf& sb);
istream& get(streambuf& sb, char_type delim);
```

Similar to the `get()` functions, except these store the result in the stream buffer `sb`.

```
istream& getline(char_type* s, streamsize n, char_type delim);
istream& getline(char_type* 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 as delimiter.

```
istream& read(char_type* s, streamsize n);
```

Extracts `n` characters from the stream into the buffer pointed to by `s`. Often used to extract binary data. Returns a reference to the stream.

```
streamsize readsome(char_type* s, streamsize n);
```

Similar as `read()`, but returns the number of extracted characters.

```
istream& ignore(streamsize n = 1, int_type delim = traits::eof());
```

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`.

```
streamsize gcount() const;
```

Returns the number of characters extracted by the last unformatted input on this object.

Input Stream Navigation

```
int_type peek();
```

Returns the next character without extracting it.

```
istream& putback(char_type 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(pos_type pos);
```

Moves to the specified position in the stream.

```
istream& seekg(off_type, ios_base::seekdir);
```

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

```
ostream& operator<<(bool n);
ostream& operator<<(short n);
ostream& operator<<(unsigned short n);
ostream& operator<<(int n);
```

```
ostream& operator<<(unsigned int n);
ostream& operator<<(long n);
ostream& operator<<(unsigned long n);
ostream& operator<<(long long n);
ostream& operator<<(unsigned long long n);
ostream& operator<<(float f);
ostream& operator<<(double f);
ostream& operator<<(long double f);
ostream& operator<<(const void* p);
```

Outputs numerical/Boolean data to the stream.

```
ostream& operator<<(ostream&, const char*);
ostream& operator<<(ostream&, const signed char*);
ostream& operator<<(ostream&, const unsigned char*);
```

Outputs character array (C-style string) data to the stream.

```
ostream& operator<<(ostream&, char);
ostream& operator<<(ostream&, signed char);
ostream& operator<<(ostream&, unsigned char);
```

Outputs character data to the stream.

```
ostream& operator<<(streambuf* sb);
```

Outputs the stream buffer `sb` to the stream.

Unformatted Output

```
ostream& put(char_type c);
```

Puts a single character onto the stream.

```
ostream& write(const char_type* 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

```
pos_type tellp();
```

Returns the current position of the stream.

```
ostream& seekp(pos_type);
```

Moves to the specified position in the stream.

```
ostream& seekp(off_type off, ios_base::seekdir dir);
```

Moves `off` characters in the specified direction `dir`.

ATOMIC OPERATIONS LIBRARY

The atomic types and operations are introduced in Chapter 22.

atomic<T>

The following operations are available for `atomic<T>`.

```
bool is_lock_free() const volatile noexcept;
bool is_lock_free() const noexcept;
```

Returns whether this `atomic<T>` has a lock-free implementation.

```
void store(T, memory_order = memory_order_seq_cst) volatile noexcept;
void store(T, memory_order = memory_order_seq_cst) noexcept;
```

Atomically stores the given value in the `atomic<T>`.

```
T load(memory_order = memory_order_seq_cst) const volatile noexcept;
T load(memory_order = memory_order_seq_cst) const noexcept;
```

Atomically returns the value of the `atomic<T>`.

```
operator T() const volatile noexcept;
operator T() const noexcept;
```

Conversion operator to type `T`.

```
T exchange(T, memory_order = memory_order_seq_cst) volatile noexcept;
T exchange(T, memory_order = memory_order_seq_cst) noexcept;
bool compare_exchange_weak(T&, T, memory_order, memory_order)
    volatile noexcept;
bool compare_exchange_weak(T&, T, memory_order, memory_order) noexcept;
bool compare_exchange_strong(T&, T, memory_order, memory_order)
    volatile noexcept;
bool compare_exchange_strong(T&, T, memory_order, memory_order) noexcept;
bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst)
    volatile noexcept;
bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst)
    noexcept;
bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst)
    volatile noexcept;
bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst)
    noexcept;
```

Performs atomic compare and exchange operations.

```
atomic() noexcept = default;
```

Default constructor.

```
constexpr atomic(T) noexcept;
```

Constructor with initial value.

```
atomic(const atomic&) = delete;
```

The copy constructor is explicitly deleted.

```
atomic& operator=(const atomic&) = delete;
atomic& operator=(const atomic&) volatile = delete;
```

The assignment operator is explicitly deleted.

atomic<integral>

A specialization for integral types, `atomic<integral>`, is available with the following extra methods compared to `atomic<T>`.

```
integral fetch_add(integral, memory_order = memory_order_seq_cst)
    volatile noexcept;
integral fetch_add(integral, memory_order = memory_order_seq_cst) noexcept;
integral fetch_sub(integral, memory_order = memory_order_seq_cst)
    volatile noexcept;
integral fetch_sub(integral, memory_order = memory_order_seq_cst) noexcept;
integral fetch_and(integral, memory_order = memory_order_seq_cst)
    volatile noexcept;
integral fetch_and(integral, memory_order = memory_order_seq_cst) noexcept;
integral fetch_or(integral, memory_order = memory_order_seq_cst)
    volatile noexcept;
integral fetch_or(integral, memory_order = memory_order_seq_cst) noexcept;
integral fetch_xor(integral, memory_order = memory_order_seq_cst)
    volatile noexcept;
integral fetch_xor(integral, memory_order = memory_order_seq_cst) noexcept;
```

Atomically performs the addition, subtraction, AND, OR and XOR operation, while fetching and returning the atomic value.

```
integral operator++(int) volatile noexcept;
integral operator++(int) noexcept;
integral operator--(int) volatile noexcept;
integral operator--(int) noexcept;
integral operator++() volatile noexcept;
integral operator++() noexcept;
integral operator--() volatile noexcept;
integral operator--() noexcept;
integral operator+=(integral) volatile noexcept;
```

```
integral operator+=(integral) noexcept;
integral operator-=(integral) volatile noexcept;
integral operator-=(integral) noexcept;
integral operator&=(integral) volatile noexcept;
integral operator&=(integral) noexcept;
integral operator|=(integral) volatile noexcept;
integral operator|=(integral) noexcept;
integral operator^=(integral) volatile noexcept;
integral operator^=(integral) noexcept;
```

Performs atomic arithmetic operations.

Atomic Integral typedefs

The standard defines the following named atomic types for integral atomic types:

NAMED ATOMIC TYPE	EQUIVALENT ATOMIC TYPE	INTEGRAL TYPE
atomic_char	atomic<char>	char
atomic_schar	atomic<signed char>	signed char
atomic_uchar	atomic<unsigned char>	unsigned char
atomic_short	atomic<short>	short
atomic_ushort	atomic<unsigned short>	unsigned short
atomic_int	atomic<int>	int
atomic_uint	atomic<unsigned int>	unsigned int
atomic_long	atomic<long>	long
atomic_ulong	atomic<unsigned long>	unsigned long
atomic_llong	atomic<long long>	long long
atomic_ullong	atomic<unsigned long long>	unsigned long long
atomic_char16_t	atomic<char16_t>	char16_t
atomic_char32_t	atomic<char32_t>	char32_t
atomic_wchar_t	atomic<wchar_t>	wchar_t

atomic<T*>

A specialization for pointer types, `atomic<T*>`, is available with the following extra methods compared to `atomic<T>`.

```
T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) volatile noexcept;
T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) noexcept;
```

Atomically performs the addition and subtraction operation, while fetching and returning the atomic value.

```
T* operator++(int) volatile noexcept;
T* operator++(int) noexcept;
T* operator--(int) volatile noexcept;
T* operator--(int) noexcept;
T* operator++() volatile noexcept;
T* operator++() noexcept;
T* operator--() volatile noexcept;
T* operator--() noexcept;
T* operator+=(ptrdiff_t) volatile noexcept;
T* operator+=(ptrdiff_t) noexcept;
T* operator-=(ptrdiff_t) volatile noexcept;
T* operator-=(ptrdiff_t) noexcept;
```

Performs atomic arithmetic operations.

Flag Type

The library provides an `atomic_flag` type that has test-and-set functionality. A flag can be set (=1) or cleared (=0).

```
bool test_and_set(memory_order = memory_order_seq_cst) volatile noexcept;
bool test_and_set(memory_order = memory_order_seq_cst) noexcept;
```

Atomically returns the current flag state, and sets the state to 1.

```
void clear(memory_order = memory_order_seq_cst) volatile noexcept;
void clear(memory_order = memory_order_seq_cst) noexcept;
```

Clears the current flag state to 0.

```
atomic_flag() noexcept = default;
```

Default constructor.

```
atomic_flag(const atomic_flag&) = delete;
```

Copy constructor is explicitly deleted.

```
atomic_flag& operator=(const atomic_flag&) = delete;
atomic_flag& operator=(const atomic_flag&) volatile = delete;
```

Assignment operator is explicitly deleted.

MULTITHREADING LIBRARY

The multithreading library is introduced in Chapter 22.

<thread>

The `thread` class has the following methods.

```
thread() noexcept;
```

Default constructor.

```
template <class F, class ...Args> explicit thread(F&& f, Args&&... args);
```

Constructor that creates a `thread` which will execute the given function `f`, with given arguments `args`.

```
~thread();
```

Destructor.

```
thread(const thread&) = delete;
```

Copy constructor is explicitly deleted.

```
thread(thread&&) noexcept;
```

Move constructor is allowed.

```
thread& operator=(const thread&) = delete;
```

Copy assignment operator is explicitly deleted.

```
thread& operator=(thread&&) noexcept;
```

Move assignment operator is allowed.

```
void swap(thread&) noexcept;
```

Swaps two `thread`s.

```
bool joinable() const noexcept;
```

Returns whether this `thread` is joinable.

```
void join();
```

Blocks the current `thread`, until the `thread` on which `join()` is called is finished.

```
void detach();
```

The calling `thread` does not wait until the `thread` on which `join()` is called is finished.

```
id get_id() const noexcept;
```

Returns an id for the `thread`.

this_thread

```
void yield() noexcept;
```

Offers the run-time library the opportunity to reschedule.

```
template <class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Sleeps until the given `time_point` is reached.

```
template <class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);
```

Sleeps for the given `duration`.

Mutual Exclusion

The multithreading library supports four kinds of mutual exclusion objects, also called mutex objects, and two different types of locks. They are explained in details in Chapter 22.

mutex and recursive_mutex

The `mutex` and `recursive_mutex` classes define:

```
constexpr mutex() noexcept;
```

Default constructor.

```
~mutex();
```

Destructor.

```
mutex(const mutex&) = delete;
mutex& operator=(const mutex&) = delete;
```

Copy constructor and assignment operator are explicitly deleted.

```
void lock();
```

Blocks the calling thread until the lock has been acquired.

```
bool try_lock();
```

Tries to acquire a lock, but does not block if another thread currently has the lock. Returns `true` if the lock has been acquired.

```
void unlock();
```

Releases the acquired lock.

timed_mutex and recursive_timed_mutex

The `timed_mutex` and `recursive_timed_mutex` classes define similar methods as the `mutex` class in addition to the following methods:

```
template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
```

Tries to acquire a lock for the given duration. If it fails, returns `false`.

```
template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Tries to acquire a lock until the given `time_point` is reached. Returns `false` if it fails.

lock_guard

```
template <class Mutex> class lock_guard;
```

The `lock_guard` class has the following methods:

```
explicit lock_guard(mutex_type& m);
lock_guard(mutex_type& m, adopt_lock_t);
```

Constructors. The first acquires a lock on the given mutex. The second assumes the calling thread already has a lock on the given mutex and simply adopts that lock.

```
~lock_guard();
```

Destructor.

```
lock_guard(lock_guard const&) = delete;
lock_guard& operator=(lock_guard const&) = delete;
```

Copy constructor and assignment operator are explicitly deleted.

unique_lock

```
template <class Mutex> class unique_lock;
```

The `unique_lock` class has the following methods:

```
unique_lock() noexcept;
```

Default constructor.

```
explicit unique_lock(mutex_type& m);
unique_lock(mutex_type& m, defer_lock_t) noexcept;
unique_lock(mutex_type& m, try_to_lock_t);
unique_lock(mutex_type& m, adopt_lock_t);
```

```
template <class Clock, class Duration>
    unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>&
        abs_time);
template <class Rep, class Period>
    unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
```

Constructors that can acquire a lock on the given mutex, adopt a lock on the mutex, or defer locking until a later point.

```
~unique_lock();
```

Destructor.

```
unique_lock(unique_lock const&) = delete;
unique_lock& operator=(unique_lock const&) = delete;
```

Copy constructor and assignment operator are explicitly deleted.

```
unique_lock(unique_lock&& u) noexcept;
unique_lock& operator=(unique_lock&& u) noexcept;
```

Move constructor and move assignment operator.

```
void lock();
```

Blocks the calling thread until a lock has been acquired.

```
bool try_lock();
```

Tries to acquire a lock, but does not block. Returns `true` when the lock has been acquired.

```
template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
```

Tries to acquire a lock for a given `duration`.

```
template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Tries to acquire a lock until `time_point` is reached.

```
void unlock();
```

Releases the lock.

```
bool owns_lock() const noexcept;
```

Returns `true` if the calling thread owns the lock.

TYPE TRAITS

Chapter 20 introduces type traits in the context of template metaprogramming. This section lists all available type traits and type traits related operations. Consult Chapter 20 for examples on how to use these.

Primary Type Categories

```
template <class T> struct is_void;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_lvalue_reference;
template <class T> struct is_rvalue_reference;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;
```

Composite Type Categories

```
template <class T> struct is_reference;
template <class T> struct is_arithmetic;
template <class T> struct is_fundamental;
template <class T> struct is_object;
template <class T> struct is_scalar;
template <class T> struct is_compound;
template <class T> struct is_member_pointer;
```

Type Properties

```
template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_trivial;
template <class T> struct is_trivially_copyable;
template <class T> struct is_standard_layout;
template <class T> struct is_pod;
template <class T> struct is_literal_type;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct is_signed;
template <class T> struct is_unsigned;
template <class T, class... Args> struct is_constructible;
template <class T> struct is_default_constructible;
template <class T> struct is_copy_constructible;
template <class T> struct is_move_constructible;
```

```
template <class T, class U> struct is_assignable;
template <class T> struct is_copy_assignable;
template <class T> struct is_move_assignable;
template <class T> struct is_destructible;
template <class T, class... Args> struct is_trivially_constructible;
template <class T> struct is_trivially_default_constructible;
template <class T> struct is_trivially_copy_constructible;
template <class T> struct is_trivially_move_constructible;
template <class T, class U> struct is_trivially_assignable;
template <class T> struct is_trivially_copy_assignable;
template <class T> struct is_trivially_move_assignable;
template <class T> struct is_trivially_destructible;
template <class T, class... Args> struct is_nothrow_constructible;
template <class T> struct is_nothrow_default_constructible;
template <class T> struct is_nothrow_copy_constructible;
template <class T> struct is_nothrow_move_constructible;
template <class T, class U> struct is_nothrow_assignable;
template <class T> struct is_nothrow_copy_assignable;
template <class T> struct is_nothrow_move_assignable;
template <class T> struct is_nothrow_destructible;
template <class T> struct has_virtual_destructor;
```

Type Property Queries

```
template <class T> struct alignment_of;
template <class T> struct rank;
template <class T, unsigned I = 0> struct extent;
```

Type Relations

```
template <class T, class U> struct is_same;
template <class Base, class Derived> struct is_base_of;
template <class From, class To> struct is_convertible;
```

const-volatile Modifications

```
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;
```

Reference Modifications

```
template <class T> struct remove_reference;
template <class T> struct add_lvalue_reference;
template <class T> struct add_rvalue_reference;
```

Sign Modifications

```
template <class T> struct make_signed;
template <class T> struct make_unsigned;
```

Array Modifications

```
template <class T> struct remove_extent;
template <class T> struct remove_all_extents;
```

Pointer Modifications

```
template <class T> struct remove_pointer;
template <class T> struct add_pointer;
```

Other Transformations

```
template <std::size_t Len, std::size_t Align> struct aligned_storage;
template <class T> struct decay;
template <bool, class T = void> struct enable_if;
template <bool, class T, class F> struct conditional;
template <class... T> struct common_type;
template <class T> struct underlying_type;
template <class> class result_of; // undefined
template <class F, class... ArgTypes> class result_of<F(ArgTypes...)>;
```

REGULAR EXPRESSION LIBRARY

The regular expression library is introduced in Chapter 14. As a reference, the following sections list the regular expression algorithms and iterators.

Function Template regex_match

```
template <class BidirectionalIterator, class Allocator, class charT, class traits>
    bool regex_match(BidirectionalIterator first,
                     BidirectionalIterator last,
                     match_results<BidirectionalIterator, Allocator>& m,
                     const basic_regex<charT, traits>& e,
                     regex_constants::match_flag_type flags =
                         regex_constants::match_default);

template <class BidirectionalIterator, class charT, class traits>
    bool regex_match(BidirectionalIterator first,
                     BidirectionalIterator last,
                     const basic_regex<charT, traits>& e,
```

```
                    regex_constants::match_flag_type flags =
                        regex_constants::match_default);

template <class charT, class Allocator, class traits>
    bool regex_match(const charT* str,
                    match_results<const charT*, Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                        regex_constants::match_default);

template <class ST, class SA, class Allocator, class charT, class traits>
    bool regex_match(const basic_string<charT, ST, SA>& s,
                    match_results<
                        typename basic_string<charT, ST, SA>::const_iterator,
                        Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                        regex_constants::match_default);

template <class charT, class traits>
    bool regex_match(const charT* str,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                        regex_constants::match_default);

template <class ST, class SA, class charT, class traits>
    bool regex_match(const basic_string<charT, ST, SA>& s,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                        regex_constants::match_default);
```

Function Template regex_search

```
template <class BidirectionalIterator, class Allocator, class charT, class traits>
    bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                     match_results<BidirectionalIterator, Allocator>& m,
                     const basic_regex<charT, traits>& e,
                     regex_constants::match_flag_type flags =
                         regex_constants::match_default);

template <class BidirectionalIterator, class charT, class traits>
    bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                     const basic_regex<charT, traits>& e,
                     regex_constants::match_flag_type flags =
                         regex_constants::match_default);

template <class charT, class Allocator, class traits>
    bool regex_search(const charT* str,
                     match_results<const charT*, Allocator>& m,
                     const basic_regex<charT, traits>& e,
                     regex_constants::match_flag_type flags =
                         regex_constants::match_default);

template <class charT, class traits>
```

```
    bool regex_search(const charT* str,
                      const basic_regex<charT, traits>& e,
                      regex_constants::match_flag_type flags =
                          regex_constants::match_default);

template <class ST, class SA, class charT, class traits>
    bool regex_search(const basic_string<charT, ST, SA>& s,
                      const basic_regex<charT, traits>& e,
                      regex_constants::match_flag_type flags =
                          regex_constants::match_default);

template <class ST, class SA, class Allocator, class charT, class traits>
    bool regex_search(const basic_string<charT, ST, SA>& s,
                      match_results<
                          typename basic_string<charT, ST, SA>::const_iterator,
                          Allocator>& m,
                      const basic_regex<charT, traits>& e,
                      regex_constants::match_flag_type flags =
                          regex_constants::match_default);
```

Function Template regex_replace

```
template <class OutputIterator, class BidirectionalIterator,
          class traits, class charT, class ST, class SA>
    OutputIterator regex_replace(
        OutputIterator out,
        BidirectionalIterator first,
        BidirectionalIterator last,
        const basic_regex<charT, traits>& e,
        const basic_string<charT, ST, SA>& fmt,
        regex_constants::match_flag_type flags =
            regex_constants::match_default);

template <class OutputIterator, class BidirectionalIterator,
          class traits, class charT>
    OutputIterator regex_replace(
        OutputIterator out,
        BidirectionalIterator first,
        BidirectionalIterator last,
        const basic_regex<charT, traits>& e,
        const charT* fmt,
        regex_constants::match_flag_type flags =
            regex_constants::match_default);

template <class traits, class charT, class ST, class SA, class FST, class FSA>
    basic_string<charT, ST, SA> regex_replace(
        const basic_string<charT, ST, SA>& s,
        const basic_regex<charT, traits>& e,
        const basic_string<charT, FST, FSA>& fmt,
        regex_constants::match_flag_type flags =
            regex_constants::match_default);

template <class traits, class charT, class ST, class SA>
    basic_string<charT, ST, SA> regex_replace(
```

```
        const basic_string<charT, ST, SA>& s,
        const basic_regex<charT, traits>& e,
        const charT* fmt,
        regex_constants::match_flag_type flags =
            regex_constants::match_default);

template <class traits, class charT, class ST, class SA>
    basic_string<charT> regex_replace(
        const charT* s,
        const basic_regex<charT, traits>& e,
        const basic_string<charT, ST, SA>& fmt,
        regex_constants::match_flag_type flags =
            regex_constants::match_default);

template <class traits, class charT>
    basic_string<charT> regex_replace(
        const charT* s,
        const basic_regex<charT, traits>& e,
        const charT* fmt,
        regex_constants::match_flag_type flags =
            regex_constants::match_default);
```

Class Template regex_iterator

```
template <class BidirectionalIterator,
    class charT = typename iterator_traits<BidirectionalIterator>::value_type,
    class traits = regex_traits<charT> >
    class regex_iterator;

typedef regex_iterator<const char*> cregex_iterator;
typedef regex_iterator<const wchar_t*> wcregex_iterator;
typedef regex_iterator<string::const_iterator> sregex_iterator;
typedef regex_iterator<wstring::const_iterator> wsregex_iterator;
```

Class Template regex_token_iterator

```
template <class BidirectionalIterator,
    class charT = typename iterator_traits<BidirectionalIterator>::value_type,
    class traits = regex_traits<charT> >
    class regex_token_iterator;

typedef regex_token_iterator<const char*> cregex_token_iterator;
typedef regex_token_iterator<const wchar_t*> wcregex_token_iterator;
typedef regex_token_iterator<string::const_iterator> sregex_token_iterator;
typedef regex_token_iterator<wstring::const_iterator> wsregex_token_iterator;
```

THE CHRONO LIBRARY

The Chrono library is introduced in Chapter 16.

The duration Class

```
template <class Rep, class Period = ratio<1>> class duration;
```

A `duration` with the specified tick period and represented by the arithmetic type `Rep`.

The `duration` class defines the following `public` members.

```
constexpr duration() = default;
```

Default constructor.

```
template <class Rep2> constexpr explicit duration(const Rep2& r);
```

Constructs a `duration` object, initialized with the given tick count.

```
template <class Rep2, class Period2>
    constexpr duration(const duration<Rep2, Period2>& d);
```

Constructs a `duration` object, initialized with the given duration, possibly converting between units.

```
~duration() = default;
```

Destructor.

```
duration(const duration&) = default;
```

Copy constructor.

```
duration& operator=(const duration&) = default;
```

Assignment operator.

```
constexpr rep count() const;
```

Returns the number of ticks for this `duration`.

```
constexpr duration operator+() const;
constexpr duration operator-() const;
duration& operator++();
duration operator++(int);
duration& operator--();
duration operator--(int);
duration& operator+=(const duration& d);
duration& operator-=(const duration& d);
duration& operator*=(const rep& rhs);
duration& operator/=(const rep& rhs);
duration& operator%=(const rep& rhs);
duration& operator%=(const duration& rhs);
```

Arithmetic operations on the `duration`.

```
static constexpr duration zero();
static constexpr duration min();
static constexpr duration max();
```

Returns a `duration` object representing zero ticks, the minimum number of ticks, or the maximum number of ticks respectively.

The time_point Class

```
template <class Clock, class Duration = typename Clock::duration>
    class time_point;
```

A `time_point` associated with the given `Clock`. `Duration` is the `duration` since the epoch of the associated `Clock`.

The `time_point` class defines the following `public` members.

```
time_point();
```

Default constructor. The `time_point` represents the epoch of the associated clock.

```
explicit time_point(const duration& d);
```

Constructor, initializing the `time_point` with time of the epoch + `d`.

```
template <class Duration2> time_point(const time_point<clock, Duration2>& t);
```

Constructor, initializing the new `time_point` with the given `time_point`, possible converting units.

```
duration time_since_epoch() const;
```

Returns the `duration` since the epoch.

```
time_point& operator+=(const duration& d);
time_point& operator-=(const duration& d);
```

Arithmetic operations on time points.

```
static constexpr time_point min();
static constexpr time_point max();
```

Returns a `time_point` object representing the minimum point in time, or the maximum point in time respectively.

Clocks

The standard defines three clocks: `system_clock`, `steady_clock`, and `high_resolution_clock`

system_clock

The system clock is a clock representing the wall clock time of the systems' real-time clock. It has the following `public` members.

```
typedef unspecified rep;
typedef ratio<unspecified, unspecified> period;
typedef chrono::duration<rep, period> duration;
typedef chrono::time_point<system_clock> time_point;
```

Specific `typedef`s for the system clock.

```
static const bool is_steady = unspecified;
```

Returns `true` if the system clock is steady.

```
static time_point now() noexcept;
```

Returns a `time_point` representing the current system time.

```
static time_t to_time_t (const time_point& t) noexcept;
static time_point from_time_t(time_t t) noexcept;
```

Two methods that provide mappings with C-style time related functions.

steady_clock

A steady clock guarantees that `time_point`s never decrease as physical time advances, and that `time_point`s advance at a steady rate relative to real time. That is, the clock may not be adjusted. It has the following `public` members.

```
typedef unspecified rep;
typedef ratio<unspecified, unspecified> period;
typedef chrono::duration<rep, period> duration;
typedef chrono::time_point<unspecified, duration> time_point;
```

Specific `typedef`s for the steady clock.

```
static const bool is_steady = true;
```

Returns `true` if the clock is steady, which is always the case for a `steady_clock`.

```
static time_point now() noexcept;
```

Returns a `time_point` representing the current steady clock time.

high_resolution_clock

A high resolution clock represents a clock with the shortest possible tick period for your system. The `high_resolution_clock` may be synonym for `system_clock` or `steady_clock`. It has the following `public` members.

```
typedef unspecified rep;
typedef ratio<unspecified, unspecified> period;
typedef chrono::duration<rep, period> duration;
typedef chrono::time_point<unspecified, duration> time_point;
```

Specific `typedef`s for the high resolution clock.

```
static const bool is_steady = unspecified;
```

Returns `true` if the clock is steady.

```
static time_point now() noexcept;
```

Returns a `time_point` representing the current high resolution clock time.