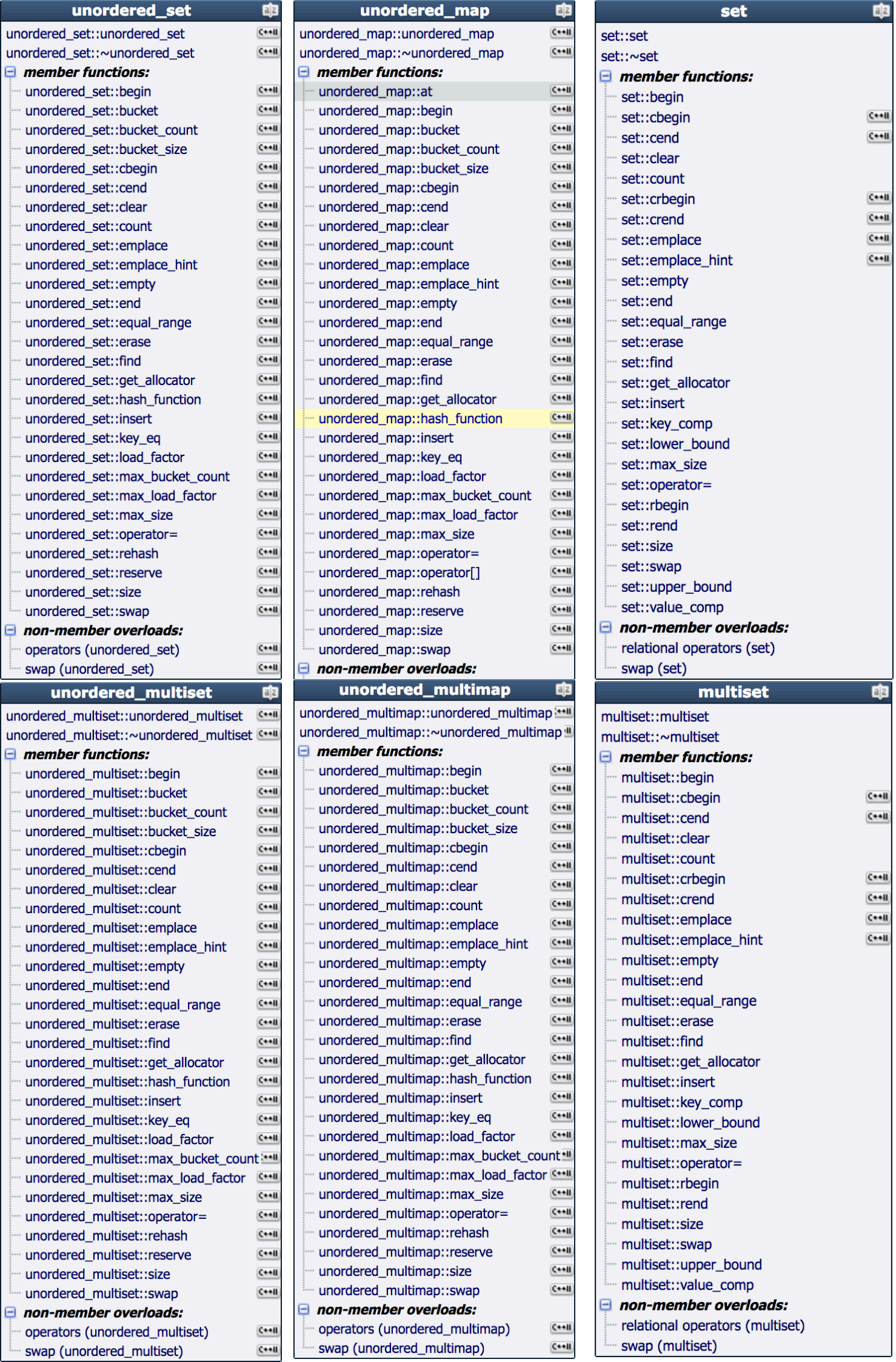

unordered_set
unordered_set::unordered_set
unordered_set::~unordered_set
member functions:
unordered_set::begin
unordered_set::bucket
unordered_set::bucket_count
unordered_set::bucket_size
unordered_set::cbegin
unordered_set::cend
unordered_set::clear
unordered_set::count
unordered_set::emplace
unordered_set::emplace_hint
unordered_set::empty
unordered_set::end
unordered_set::equal_range
unordered_set::erase
unordered_set::find
unordered_set::get_allocator
unordered_set::hash_function
unordered_set::insert
unordered_set::key_eq
unordered_set::load_factor
unordered_set::max_bucket_count
unordered_set::max_load_factor
unordered_set::max_size
unordered_set::operator=
unordered_set::rehash
unordered_set::reserve
unordered_set::size
unordered_set::swap
non-member overloads:
operators (unordered_set)
swap (unordered_set)
unordered_map
unordered_map::unordered_map
unordered_map::~unordered_map
member functions:
unordered_map::at
unordered_map::begin
unordered_map::bucket
unordered_map::bucket_count
unordered_map::bucket_size
unordered_map::cbegin
unordered_map::cend
unordered_map::clear
unordered_map::count
unordered_map::emplace
unordered_map::emplace_hint
unordered_map::empty
unordered_map::end
unordered_map::equal_range
unordered_map::erase
unordered_map::find
unordered_map::get_allocator
unordered_map::hash_function
unordered_map::insert
unordered_map::key_eq
unordered_map::load_factor
unordered_map::max_bucket_count
unordered_map::max_load_factor
unordered_map::max_size
unordered_map::operator=
unordered_map::operator[]
unordered_map::rehash
unordered_map::reserve
unordered_map::size
unordered_map::swap
non-member overloads:
set
set::set
set::~set
member functions:
set::begin
set::cbegin
set::cend
set::clear
set::count
set::crbegin
set::crend
set::emplace
set::emplace_hint
set::empty
set::end
set::equal_range
set::erase
set::find
set::get_allocator
set::insert
set::key_comp
set::lower_bound
set::max_size
set::operator=
set::rbegin
set::rend
set::size
set::swap
set::upper_bound
set::value_comp
non-member overloads:
relational operators (set)
swap (set)
unordered_multiset
unordered_multiset::unordered_multiset
unordered_multiset::~unordered_multiset
member functions:
unordered_multiset::begin
unordered_multiset::bucket
unordered_multiset::bucket_count
unordered_multiset::bucket_size
unordered_multiset::cbegin
unordered_multiset::cend
unordered_multiset::clear
unordered_multiset::count
unordered_multiset::emplace
unordered_multiset::emplace_hint
unordered_multiset::empty
unordered_multiset::end
unordered_multiset::equal_range
unordered_multiset::erase
unordered_multiset::find
unordered_multiset::get_allocator
unordered_multiset::hash_function
unordered_multiset::insert
unordered_multiset::key_eq
unordered_multiset::load_factor
unordered_multiset::max_bucket_count
unordered_multiset::max_load_factor
unordered_multiset::max_size
unordered_multiset::operator=
unordered_multiset::rehash
unordered_multiset::reserve
unordered_multiset::size
unordered_multiset::swap
non-member overloads:
operators (unordered_multiset)
swap (unordered_multiset)
unordered_multimap
unordered_multimap::unordered_multimap
unordered_multimap::~unordered_multimap
member functions:
unordered_multimap::begin
unordered_multimap::bucket
unordered_multimap::bucket_count
unordered_multimap::bucket_size
unordered_multimap::cbegin
unordered_multimap::cend
unordered_multimap::clear
unordered_multimap::count
unordered_multimap::emplace
unordered_multimap::emplace_hint
unordered_multimap::empty
unordered_multimap::end
unordered_multimap::equal_range
unordered_multimap::erase
unordered_multimap::find
unordered_multimap::get_allocator
unordered_multimap::hash_function
unordered_multimap::insert
unordered_multimap::key_eq
unordered_multimap::load_factor
unordered_multimap::max_bucket_count
unordered_multimap::max_load_factor
unordered_multimap::max_size
unordered_multimap::operator=
unordered_multimap::rehash
unordered_multimap::reserve
unordered_multimap::size
unordered_multimap::swap
non-member overloads:
operators (unordered_multimap)
swap (unordered_multimap)
multiset
multiset::multiset
multiset::~multiset
member functions:
multiset::begin
multiset::cbegin
multiset::cend
multiset::clear
multiset::count
multiset::crbegin
multiset::crend
multiset::emplace
multiset::emplace_hint
multiset::empty
multiset::end
multiset::equal_range
multiset::erase
multiset::find
multiset::get_allocator
multiset::insert
multiset::key_comp
multiset::lower_bound
multiset::max_size
multiset::operator=
multiset::rbegin
multiset::rend
multiset::size
multiset::swap
multiset::upper_bound
multiset::value_comp
non-member overloads:
relational operators (multiset)
swap (multiset)

map

- map::map
- map::~map
- ***member functions:***
  - map::at C++11
  - map::begin
  - map::cbegin C++11
  - map::cend C++11
  - map::clear
  - map::count
  - map::crbegin C++11
  - map::crend C++11
  - map::emplace C++11
  - map::emplace_hint C++11
  - map::empty
  - map::end
  - map::equal_range
  - map::erase
  - map::find
  - map::get_allocator
  - map::insert
  - map::key_comp
  - map::lower_bound
  - map::max_size
  - map::operator=
  - map::operator[]
  - map::rbegin
  - map::rend
  - map::size
  - map::swap
  - map::upper_bound
  - map::value_comp
- ***non-member overloads:***
  - relational operators (map)
  - swap (map)

multimap

- multimap::multimap
- multimap::~multimap
- ***member functions:***
  - multimap::begin
  - multimap::cbegin C++11
  - multimap::cend C++11
  - multimap::clear
  - multimap::count
  - multimap::crbegin C++11
  - multimap::crend C++11
  - multimap::emplace C++11
  - multimap::emplace_hint C++11
  - multimap::empty
  - multimap::end
  - multimap::equal_range
  - multimap::erase
  - multimap::find
  - multimap::get_allocator
  - multimap::insert
  - multimap::key_comp
  - multimap::lower_bound
  - multimap::max_size
  - multimap::operator=
  - multimap::rbegin
  - multimap::rend
  - multimap::size
  - multimap::swap
  - multimap::upper_bound
  - multimap::value_comp
- ***non-member overloads:***
  - relational operators (multimap)
  - swap (multimap)

stack

- stack::stack
- ***member functions:***
  - stack::emplace C++11
  - stack::empty
  - stack::pop
  - stack::push
  - stack::size
  - stack::swap C++11
  - stack::top
- ***non-member overloads:***
  - relational operators (stack)
  - swap (stack) C++11
- ***non-member specializations:***
  - uses_allocator<stack> C++11

list

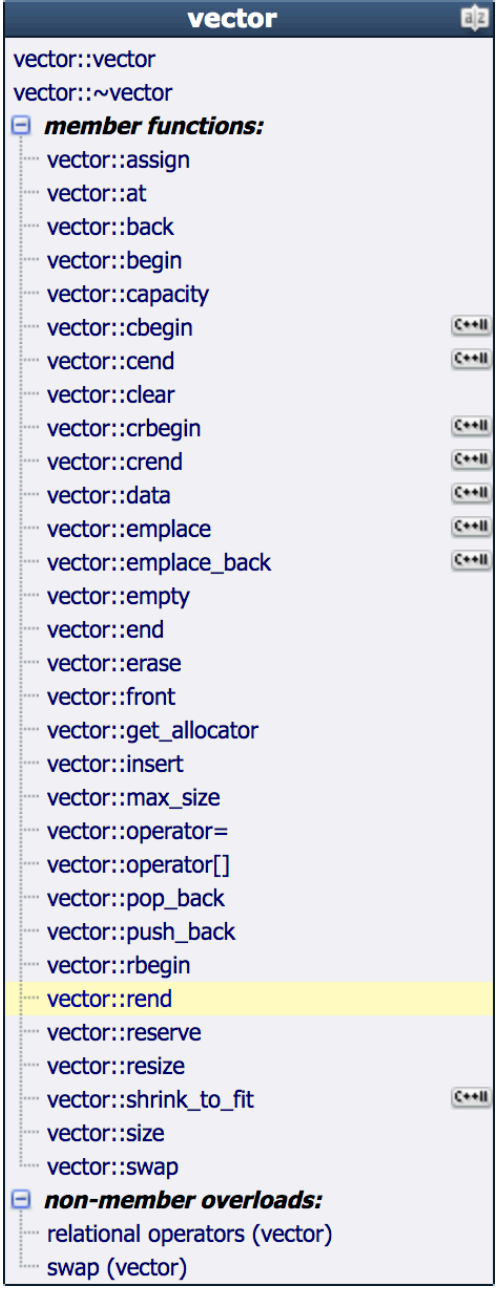
- list::list
- list::~list
- ***member functions:***
  - list::assign
  - list::back
  - list::begin
  - list::cbegin C++11
  - list::cend C++11
  - list::clear
  - list::crbegin C++11
  - list::crend C++11
  - list::emplace C++11
  - list::emplace_back C++11
  - list::emplace_front C++11
  - list::empty
  - list::end
  - list::erase
  - list::front
  - list::get_allocator
  - list::insert
  - list::max_size
  - list::merge
  - list::operator=
  - list::pop_back
  - list::pop_front
  - list::push_back
  - list::push_front
  - list::rbegin
  - list::remove
  - list::remove_if
  - list::rend
  - list::resize
  - list::reverse
  - list::size
  - list::sort
  - list::splice
  - list::swap
  - list::unique
- ***non-member overloads:***
  - relational operators (list)
  - swap (list)

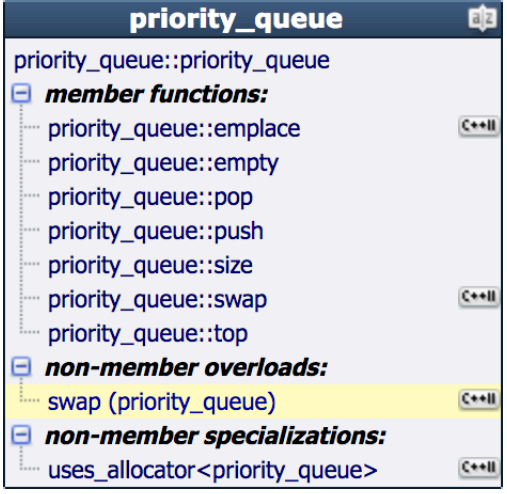
deque

- deque::deque
- deque::~deque
- ***member functions:***
  - deque::assign
  - deque::at
  - deque::back
  - deque::begin
  - deque::cbegin C++11
  - deque::cend C++11
  - deque::clear
  - deque::crbegin C++11
  - deque::crend C++11
  - deque::emplace C++11
  - deque::emplace_back C++11
  - deque::emplace_front C++11
  - deque::empty
  - deque::end
  - deque::erase
  - deque::front
  - deque::get_allocator
  - deque::insert
  - deque::max_size
  - deque::operator=
  - deque::operator[]
  - deque::pop_back
  - deque::pop_front
  - deque::push_back
  - deque::push_front
  - deque::rbegin
  - deque::rend
  - deque::resize
  - deque::shrink_to_fit C++11
  - deque::size
  - deque::swap
- ***non-member overloads:***
  - relational operators (deque)
  - swap (deque)

queue

- queue::queue
- ***member functions:***
  - queue::back
  - queue::emplace C++11
  - queue::empty
  - queue::front
  - queue::pop
  - queue::push
  - queue::size
  - queue::swap C++11
- ***non-member overloads:***
  - relational operators (queue)
  - swap (queue) C++11
- ***non-member specializations:***
  - uses_allocator<queue> C++11





std::unordered_multiset

<unordered_set>

```
template < class Key,                          // unordered_multiset::key_type/value_type
           class Hash = hash<Key>,             // unordered_multiset::hasher
           class Pred = equal_to<Key>,         // unordered_multiset::key_equal
           class Alloc = allocator<Key>        // unordered_multiset::allocator_type
           > class unordered_multiset;
```

Unordered Multiset

Unordered multisets are containers that store elements in no particular order, allowing fast retrieval of individual elements based on their value, much like unordered_set containers, but allowing different elements to have equivalent values.

In an `unordered_multiset`, the value of an element is at the same time its *key*, used to identify it. Keys are immutable, therefore, the elements in an `unordered_multiset` cannot be modified once in the container - they can be inserted and removed, though.

Internally, the elements in the `unordered_multiset` are not sorted in any particular, but organized into *buckets* depending on their hash values to allow for fast access to individual elements directly by their *values* (with a constant average time complexity on average).

Elements with equivalent values are grouped together in the same bucket and in such a way that an iterator (see equal_range) can iterate through all of them.

Iterators in the container are at least forward iterators.

Notice that this container is not defined in its own header, but shares header `<unordered_set>` with unordered_set.

std::unordered_map

<unordered_map>

```
template < class Key,                                    // unordered_map::key_type
           class T,                                      // unordered_map::mapped_type
           class Hash = hash<Key>,                       // unordered_map::hasher
           class Pred = equal_to<Key>,                   // unordered_map::key_equal
           class Alloc = allocator< pair<const Key,T> >  // unordered_map::allocator_type
           > class unordered_map;
```

Unordered Map

Unordered maps are associative containers that store elements formed by the combination of a *key value* and a *mapped value*, and which allows for fast retrieval of individual elements based on their keys.

In an `unordered_map`, the *key value* is generally used to uniquely identify the element, while the *mapped value* is an object with the content associated to this *key*. Types of *key* and *mapped* value may differ.

Internally, the elements in the `unordered_map` are not sorted in any particular order with respect to either their *key* or *mapped* values, but organized into *buckets* depending on their hash values to allow for fast access to individual elements directly by their *key values* (with a constant average time complexity on average).

`unordered_map` containers are faster than `map` containers to access individual elements by their *key*, although they are generally less efficient for range iteration through a subset of their elements.

Unordered maps implement the direct access operator (`operator[]`) which allows for direct access of the *mapped value* using its *key value* as argument.

Iterators in the container are at least forward iterators.

std::set

<set>

```
template < class T,                        // set::key_type/value_type
           class Compare = less<T>,        // set::key_compare/value_compare
           class Alloc = allocator<T>      // set::allocator_type
           > class set;
```

Set

Sets are containers that store unique elements following a specific order.

In a `set`, the value of an element also identifies it (the value is itself the *key*, of type `T`), and each value must be unique. The value of the elements in a `set` cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

Internally, the elements in a `set` are always sorted following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type `Compare`).

`set` containers are generally slower than unordered_set containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

Sets are typically implemented as *binary search trees*.

std::multiset

<set>

```
template < class T,                        // multiset::key_type/value_type
           class Compare = less<T>,        // multiset::key_compare/value_compare
           class Alloc = allocator<T> >    // multiset::allocator_type
           > class multiset;
```

Multiple-key set

Multisets are containers that store elements following a specific order, and where multiple elements can have equivalent values.

In a `multiset`, the value of an element also identifies it (the value is itself the *key*, of type `T`). The value of the elements in a `multiset` cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

Internally, the elements in a `multiset` are always sorted following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type `Compare`).

`multiset` containers are generally slower than unordered_multiset containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

Multisets are typically implemented as *binary search trees*.

std::map

<map>

```
template < class Key,                                    // map::key_type
           class T,                                      // map::mapped_type
           class Compare = less<Key>,                    // map::key_compare
           class Alloc = allocator<pair<const Key,T> >   // map::allocator_type
           > class map;
```

Map

Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.

In a `map`, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type `value_type`, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

Internally, the elements in a `map` are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type `Compare`).

`map` containers are generally slower than unordered_map containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

The mapped values in a map can be accessed directly by their corresponding key using the *bracket operator* ((operator[]).

Maps are typically implemented as *binary search trees*.

std::multimap

<map>

```
template < class Key,                                    // multimap::key_type
           class T,                                      // multimap::mapped_type
           class Compare = less<Key>,                    // multimap::key_compare
           class Alloc = allocator<pair<const Key,T> >   // multimap::allocator_type
           > class multimap;
```

Multiple-key map

Multimaps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order, and where multiple elements can have equivalent keys.

In a `multimap`, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type `value_type`, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

Internally, the elements in a `multimap` are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type `Compare`).

`multimap` containers are generally slower than unordered_multimap containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

Multimaps are typically implemented as *binary search trees*.

std::unordered_multimap

<unordered_map>

```
template < class Key,                                    // unordered_multimap::key_type
           class T,                                      // unordered_multimap::mapped_type
           class Hash = hash<Key>,                       // unordered_multimap::hasher
           class Pred = equal_to<Key>,                   // unordered_multimap::key_equal
           class Alloc = allocator< pair<const Key,T> >  // unordered_multimap::allocator_type
           > class unordered_multimap;
```

Unordered Multimap

Unordered multimaps are associative containers that store elements formed by the combination of a *key value* and a *mapped value*, much like unordered_map containers, but allowing different elements to have equivalent keys.

In an `unordered_multimap`, the *key value* is generally used to uniquely identify the element, while the *mapped value* is an object with the content associated to this *key*. Types of *key* and *mapped* value may differ.

Internally, the elements in the `unordered_multimap` are not sorted in any particular order with respect to either their *key* or *mapped* values, but organized into *buckets* depending on their hash values to allow for fast access to individual elements directly by their *key values* (with a constant average time complexity on average).

Elements with equivalent keys are grouped together in the same bucket and in such a way that an iterator (see equal_range) can iterate through all of them.

Iterators in the container are at least forward iterators.

Notice that this container is not defined in its own header, but shares header `<unordered_map>` with unordered_map.

std::unordered_set

<unordered_set>

```
template < class Key,                        // unordered_set::key_type/value_type
           class Hash = hash<Key>,           // unordered_set::hasher
           class Pred = equal_to<Key>,       // unordered_set::key_equal
           class Alloc = allocator<Key>      // unordered_set::allocator_type
           > class unordered_set;
```

Unordered Set

Unordered sets are containers that store unique elements in no particular order, and which allow for fast retrieval of individual elements based on their value.

In an `unordered_set`, the value of an element is at the same time its *key*, that identifies it uniquely. Keys are immutable, therefore, the elements in an `unordered_set` cannot be modified once in the container - they can be inserted and removed, though.

Internally, the elements in the `unordered_set` are not sorted in any particular order, but organized into *buckets* depending on their hash values to allow for fast access to individual elements directly by their *values* (with a constant average time complexity on average).

`unordered_set` containers are faster than set containers to access individual elements by their *key*, although they are generally less efficient for range iteration through a subset of their elements.

Iterators in the container are at least forward iterators.

std::deque


```
template < class T, class Alloc = allocator<T> > class deque;
```

Double ended queue

deque (usually pronounced like *"deck"*) is an irregular acronym of **d**ouble-**e**nded **que**ue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

Specific libraries may implement *deques* in different ways, generally as some form of dynamic array. But in any case, they allow for the individual elements to be accessed directly through random access iterators, with storage handled automatically by expanding and contracting the container as needed.

Therefore, they provide a functionality similar to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end. But, unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations: accessing elements in a `deque` by offsetting a pointer to another element causes *undefined behavior*.

Both vectors and deques provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a deque can be scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time and with a uniform sequential interface (through iterators). Therefore, deques are a little more complex internally than vectors, but this allows them to grow more efficiently under certain circumstances, especially with very long sequences, where reallocations become more expensive.

For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, deques perform worse and have less consistent iterators and references than lists and forward lists.

std::binary_search

<algorithm>

```
template <class ForwardIterator, class T>
  bool binary_search (ForwardIterator first, ForwardIterator last,
                      const T& val);
```
default (1)

```
template <class ForwardIterator, class T, class Compare>
  bool binary_search (ForwardIterator first, ForwardIterator last,
                      const T& val, Compare comp);
```
custom (2)

std::copy

<algorithm>

```
template <class InputIterator, class OutputIterator>
  OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result);
```

Copy range of elements

Copies the elements in the range `[first,last)` into the range beginning at *result*.

The function returns an iterator to the end of the destination range (which points to the element following the last element copied).

The ranges shall not overlap in such a way that *result* points to an element in the range *[first,last)*. For such cases, see `copy_backward`.

std::copy

<algorithm>

```
template <class InputIterator, class OutputIterator>
  OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result);
```

Copy range of elements

Copies the elements in the range `[first,last)` into the range beginning at *result*.

The function returns an iterator to the end of the destination range (which points to the element following the last element copied).

The ranges shall not overlap in such a way that *result* points to an element in the range *[first,last)*. For such cases, see `copy_backward`.

std::equal

<algorithm>

equality (1)

```
template <class InputIterator1, class InputIterator2>
  bool equal (InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2);
```

predicate (2)

```
template <class InputIterator1, class InputIterator2, class BinaryPredicate>
  bool equal (InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, BinaryPredicate pred);
```

Test whether the elements in two ranges are equal

Compares the elements in the range `[first1,last1)` with those in the range beginning at *first2*, and returns `true` if all of the elements in both ranges match.

The elements are compared using `operator==` (or *pred*, in version *(2)*).

std::equal_range

<algorithm>

default (1)

```
template <class ForwardIterator, class T>
  pair<ForwardIterator,ForwardIterator>
    equal_range (ForwardIterator first, ForwardIterator last, const T& val);
```

custom (2)

```
template <class ForwardIterator, class T, class Compare>
  pair<ForwardIterator,ForwardIterator>
    equal_range (ForwardIterator first, ForwardIterator last, const T& val,
                 Compare comp);
```

Get subrange of equal elements

Returns the bounds of the subrange that includes all the elements of the range `[first,last)` with values equivalent to *val*.

The elements are compared using `operator<` for the first version, and *comp* for the second. Two elements, `a` and `b` are considered equivalent if `(!(a<b) && !(b<a))` or if `(!comp(a,b) && !comp(b,a))`.

The elements in the range shall already be sorted according to this same criterion (`operator<` or *comp*), or at least partitioned with respect to *val*.

If *val* is not equivalent to any value in the range, the subrange returned has a length of zero, with both iterators pointing to the nearest value greater than *val*, if any, or to *last*, if *val* compares greater than all the elements in the range.

Return value

A `pair` object, whose member `pair::first` is an iterator to the lower bound of the subrange of equivalent values, and `pair::second` its upper bound.
The values are the same as those that would be returned by functions `lower_bound` and `upper_bound` respectively.

std::find

<algorithm>

```
template <class InputIterator, class T>
   InputIterator find (InputIterator first, InputIterator last, const T& val);
```

std::iter_swap

<algorithm>

```
template <class ForwardIterator1, class ForwardIterator2>
  void iter_swap (ForwardIterator1 a, ForwardIterator2 b);
```

Exchange values of objects pointed to by two iterators

Swaps the elements pointed to by *a* and *b*.

std::find_end

<algorithm>

```
equality (1)    template <class ForwardIterator1, class ForwardIterator2>
                   ForwardIterator1 find_end (ForwardIterator1 first1, ForwardIterator1 last1,
                                              ForwardIterator2 first2, ForwardIterator2 last2);
predicate (2)   template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
                   ForwardIterator1 find_end (ForwardIterator1 first1, ForwardIterator1 last1,
                                              ForwardIterator2 first2, ForwardIterator2 last2,
                                              BinaryPredicate pred);
```

Find last subsequence in range

Searches the range `[first1,last1)` for the last occurrence of the sequence defined by `[first2,last2)`, and returns an iterator to its first element, or *last1* if no occurrences are found.

The elements in both ranges are compared sequentially using `operator==` (or *pred*, in version *(2)*): A subsequence of `[first1,last1)` is considered a match only when this is `true` for **all** the elements of `[first2,last2)`.

This function returns the last of such occurrences. For an algorithm that returns the first instead, see `search`.

Return value

An iterator to the first element of the last occurrence of `[first2,last2)` in `[first1,last1)`.
If the sequence is not found, the function returns *last1*.

std::merge

<algorithm>

```
default (1)     template <class InputIterator1, class InputIterator2, class OutputIterator>
                  OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                                        InputIterator2 first2, InputIterator2 last2,
                                        OutputIterator result);
custom (2)      template <class InputIterator1, class InputIterator2,
                          class OutputIterator, class Compare>
                  OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                                        InputIterator2 first2, InputIterator2 last2,
                                        OutputIterator result, Compare comp);
```

Merge sorted ranges

Combines the elements in the sorted ranges `[first1,last1)` and `[first2,last2)`, into a new range beginning at *result* with all its elements sorted.

The elements are compared using `operator<` for the first version, and *comp* for the second. The elements in both ranges shall already be ordered according to this same criterion (`operator<` or *comp*). The resulting range is also sorted according to this.

std::includes

<algorithm>

```
template <class InputIterator1, class InputIterator2>
  bool includes ( InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2 );

template <class InputIterator1, class InputIterator2, class Compare>
  bool includes ( InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2, Compare comp );
```

Test whether sorted range includes another sorted range

Returns true if the sorted range `[first1,last1)` contains all the elements in the sorted range `[first2,last2)`.

The elements are compared using `operator<` for the first version, and *comp* for the second. Two elements, `a` and `b` are considered equivalent if `(!(a<b) && !(b<a))` or if `(!comp(a,b) && !comp(b,a))`.

The elements in the range shall already be ordered according to this same criterion (`operator<` or *comp*).

Complexity

Up to linear in twice the distances in both ranges: Performs up to `2*(count1+count2)-1` comparisons (where *countX* is the distance between *firstX* and *lastX*).

std::is_heap

<algorithm>

```
default (1)  template <class RandomAccessIterator>
               bool is_heap (RandomAccessIterator first, RandomAccessIterator last);
custom (2)   template <class RandomAccessIterator, class Compare>
               bool is_heap (RandomAccessIterator first, RandomAccessIterator last,
                             Compare comp);
```

Test if range is heap

Returns `true` if the range `[first,last)` forms a *heap*, as if constructed with `make_heap`.

The elements are compared using `operator<` for the first version, and *comp* for the second.

Complexity

Up to linear in one less than the distance between *first* and *last*: Compares pairs of elements until a mismatch is found.

std::is_heap_until

<algorithm>

```
default (1)  template <class RandomAccessIterator>
               RandomAccessIterator is_heap_until (RandomAccessIterator first,
                                                   RandomAccessIterator last);
custom (2)   template <class RandomAccessIterator, class Compare>
               RandomAccessIterator is_heap_until (RandomAccessIterator first,
                                                   RandomAccessIterator last
                                                   Compare comp);
```

Find first element not in heap order

Returns an iterator to the first element in the range `[first,last)` which is not in a valid position if the range is considered a heap (as if constructed with `make_heap`).

The range between *first* and the iterator returned is a heap.

If the entire range is a valid heap, the function returns *last*.

The elements are compared using `operator<` for the first version, and *comp* for the second.

Complexity

Up to linear in the distance between *first* and *last*: Compares elements until a mismatch is found.

std::lower_bound

<algorithm>

```
default (1)  template <class ForwardIterator, class T>
               ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last,
                                            const T& val);
custom (2)   template <class ForwardIterator, class T, class Compare>
               ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last,
                                            const T& val, Compare comp);
```

Return iterator to lower bound

Returns an iterator pointing to the first element in the range `[first,last)` which does not compare less than *val*.

The elements are compared using `operator<` for the first version, and *comp* for the second. The elements in the range shall already be sorted according to this same criterion (`operator<` or *comp*), or at least partitioned with respect to *val*.

Complexity

On average, logarithmic in the distance between *first* and *last*: Performs approximately $\log_2(N)+1$ element comparisons (where *N* is this distance).
On *non-random-access iterators*, the iterator advances produce themselves an additional linear complexity in *N* on average.

std::reverse

<algorithm>

```
template <class BidirectionalIterator>
  void reverse (BidirectionalIterator first, BidirectionalIterator last);
```

Reverse range

Reverses the order of the elements in the range `[first,last)`.

The function calls `iter_swap` to swap the elements to their new locations.

std::make_heap

<algorithm>

```
default (1)  template <class RandomAccessIterator>
               void make_heap (RandomAccessIterator first, RandomAccessIterator last);
custom (2)   template <class RandomAccessIterator, class Compare>
               void make_heap (RandomAccessIterator first, RandomAccessIterator last,
                               Compare comp );
```

Make heap from range

Rearranges the elements in the range `[first,last)` in such a way that they form a *heap*.

A *heap* is a way to organize the elements of a range that allows for fast retrieval of the element with the highest value at any moment (with pop_heap), even repeatedly, while allowing for fast insertion of new elements (with push_heap).

The element with the highest value is always pointed by *first*. The order of the other elements depends on the particular implementation, but it is consistent throughout all heap-related functions of this header.

The elements are compared using `operator<` (for the first version), or *comp* (for the second): The element with the highest value is an element for which this would return `false` when compared to every other element in the range.

The standard container adaptor `priority_queue` calls `make_heap`, `push_heap` and `pop_heap` automatically to maintain heap properties for a container.

Complexity

Up to linear in three times the distance between *first* and *last*: Compares elements and potentially swaps (or moves) them until rearranged as a heap.

std::max_element

<algorithm>

```
default (1)  template <class ForwardIterator>
               ForwardIterator max_element (ForwardIterator first, ForwardIterator last);
custom (2)   template <class ForwardIterator, class Compare>
               ForwardIterator max_element (ForwardIterator first, ForwardIterator last,
                                            Compare comp);
```

Return largest element in range

Returns an iterator pointing to the element with the largest value in the range `[first,last)`.

std::next_permutation

<algorithm>

```
default (1)  template <class BidirectionalIterator>
               bool next_permutation (BidirectionalIterator first,
                                      BidirectionalIterator last);
custom (2)   template <class BidirectionalIterator, class Compare>
               bool next_permutation (BidirectionalIterator first,
                                      BidirectionalIterator last, Compare comp);
```

Transform range to next permutation

Rearranges the elements in the range `[first,last)` into the next *lexicographically greater* permutation.

A *permutation* is each one of the `N!` possible arrangements the elements can take (where *N* is the number of elements in the range). Different permutations can be ordered according to how they compare *lexicographicaly* to each other; The first such-sorted possible permutation (the one that would compare *lexicographically smaller* to all other permutations) is the one which has all its elements sorted in ascending order, and the largest has all its elements sorted in descending order.

The comparisons of individual elements are performed using either `operator<` for the first version, or *comp* for the second.

If the function can determine the next higher permutation, it rearranges the elements as such and returns `true`. If that was not possible (because it is already at the largest possible permutation), it rearranges the elements according to the first permutation (sorted in ascending order) and returns `false`.

Return value

`true` if the function could rearrange the object as a lexicographicaly greater permutation.
Otherwise, the function returns `false` to indicate that the arrangement is not greater than the previous, but the lowest possible (sorted in ascending order).

Complexity

Up to linear in half the distance between *first* and *last* (in terms of actual swaps).

std::upper_bound

<algorithm>

```
default (1)  template <class ForwardIterator, class T>
               ForwardIterator upper_bound (ForwardIterator first, ForwardIterator last,
                                            const T& val);
custom (2)   template <class ForwardIterator, class T, class Compare>
               ForwardIterator upper_bound (ForwardIterator first, ForwardIterator last,
                                            const T& val, Compare comp);
```

Return iterator to upper bound

Returns an iterator pointing to the first element in the range `[first,last)` which compares greater than *val*.

The elements are compared using `operator<` for the first version, and *comp* for the second. The elements in the range shall already be sorted according to this same criterion (`operator<` or *comp*), or at least partitioned with respect to *val*.

std::pop_heap

<algorithm>

```
default (1)  template <class RandomAccessIterator>
               void pop_heap (RandomAccessIterator first, RandomAccessIterator last);
custom (2)   template <class RandomAccessIterator, class Compare>
               void pop_heap (RandomAccessIterator first, RandomAccessIterator last,
                              Compare comp);
```

Pop element from heap range

Rearranges the elements in the heap range `[first,last)` in such a way that the part considered a heap is shortened by one: The element with the highest value is moved to `(last-1)`.

While the element with the highest value is moved from *first* to `(last-1)` (which now is out of the heap), the other elements are reorganized in such a way that the range `[first,last-1)` preserves the properties of a heap.

A range can be organized into a heap by calling `make_heap`. After that, its heap properties are preserved if elements are added and removed from it using `push_heap` and `pop_heap`, respectively.

Complexity

Up to twice logarithmic in the distance between *first* and *last*: Compares elements and potentially swaps (or moves) them until rearranged as a shorter heap.

std::push_heap

<algorithm>

```
default (1)  template <class RandomAccessIterator>
               void push_heap (RandomAccessIterator first, RandomAccessIterator last);
custom (2)   template <class RandomAccessIterator, class Compare>
               void push_heap (RandomAccessIterator first, RandomAccessIterator last,
                               Compare comp);
```

Push element into heap range

Given a heap in the range `[first,last-1)`, this function extends the range considered a heap to `[first,last)` by placing the value in `(last-1)` into its corresponding location within it.

A range can be organized into a heap by calling `make_heap`. After that, its heap properties are preserved if elements are added and removed from it using `push_heap` and `pop_heap`, respectively.

Complexity

Up to logarithmic in the distance between *first* and *last*: Compares elements and potentially swaps (or moves) them until rearranged as a longer heap.

std::partition

<algorithm>

C++98 | C++11

```
template <class ForwardIterator, class UnaryPredicate>
  ForwardIterator partition (ForwardIterator first,
                             ForwardIterator last, UnaryPredicate pred);
```

Partition range in two

Rearranges the elements from the range `[first,last)`, in such a way that all the elements for which *pred* returns `true` precede all those for which it returns `false`. The iterator returned points to the first element of the second group.

The relative ordering within each group is not necessarily the same as before the call. See `stable_partition` for a function with a similar behavior but with stable ordering within each group.

Complexity

Linear in the distance between *first* and *last*: Applies *pred* to each element, and possibly swaps some of them (if the iterator type is a bidirectional, at most half that many swaps, otherwise at most that many).

std::search

<algorithm>

```
template <class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2, ForwardIterator2 last2);
```
equality (1)

```
template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2, ForwardIterator2 last2,
                           BinaryPredicate pred);
```
predicate (2)

Search range for subsequence

Searches the range `[first1,last1)` for the first occurrence of the sequence defined by `[first2,last2)`, and returns an iterator to its first element, or *last1* if no occurrences are found.

The elements in both ranges are compared sequentially using `operator==` (or *pred*, in version *(2)*): A subsequence of `[first1,last1)` is considered a match only when this is `true` for **all** the elements of `[first2,last2)`.

This function returns the first of such occurrences. For an algorithm that returns the last instead, see `find_end`.

Complexity

Up to linear in `count1*count2` (where *countX* is the distance between *firstX* and *lastX*): Compares elements until a matching subsequence is found.

std::rotate

<algorithm>

C++98 | C++11

```
template <class ForwardIterator>
  ForwardIterator rotate (ForwardIterator first, ForwardIterator middle,
                          ForwardIterator last);
```

Rotate left the elements in range

Rotates the order of the elements in the range `[first,last)`, in such a way that the element pointed by *middle* becomes the new first element.

std::reverse_copy

<algorithm>

```
template <class BidirectionalIterator, class OutputIterator>
  OutputIterator reverse_copy (BidirectionalIterator first,
                               BidirectionalIterator last, OutputIterator result);
```

Copy range reversed

Copies the elements in the range `[first,last)` to the range beginning at *result*, but in reverse order.

std::remove

<algorithm>

```
template <class ForwardIterator, class T>
  ForwardIterator remove (ForwardIterator first, ForwardIterator last, const T& val);
```

Remove value from range

[Note: This is the reference for algorithm `remove`. See `remove` for `<cstdio>`'s remove.]

Transforms the range `[first,last)` into a range with all the elements that compare equal to *val* removed, and returns an iterator to the new end of that range.

The function cannot alter the properties of the object containing the range of elements (i.e., it cannot alter the size of an array or a container): The removal is done by replacing the elements that compare equal to *val* by the next element that does not, and signaling the new size of the shortened range by returning an iterator to the element that should be considered its new *past-the-end* element.

The relative order of the elements not removed is preserved, while the elements between the returned iterator and *last* are left in a valid but unspecified state.

The function uses `operator==` to compare the individual elements to *val*.

std::partial_sort

<algorithm>

```
default (1)   template <class RandomAccessIterator>
                void partial_sort (RandomAccessIterator first, RandomAccessIterator middle,
                                   RandomAccessIterator last);
custom (2)    template <class RandomAccessIterator, class Compare>
                void partial_sort (RandomAccessIterator first, RandomAccessIterator middle,
                                   RandomAccessIterator last, Compare comp);
```

Partially sort elements in range

Rearranges the elements in the range `[first,last)`, in such a way that the elements before *middle* are the smallest elements in the entire range and are sorted in ascending order, while the remaining elements are left without any specific order.

The elements are compared using `operator<` for the first version, and *comp* for the second.

std::nth_element

<algorithm>

```
default (1)   template <class RandomAccessIterator>
                void nth_element (RandomAccessIterator first, RandomAccessIterator nth,
                                  RandomAccessIterator last);
custom (2)    template <class RandomAccessIterator, class Compare>
                void nth_element (RandomAccessIterator first, RandomAccessIterator nth,
                                  RandomAccessIterator last, Compare comp);
```

Sort element in range

Rearranges the elements in the range `[first,last)`, in such a way that the element at the *nth* position is the element that would be in that position in a sorted sequence.

The other elements are left without any specific order, except that none of the elements preceding *nth* are greater than it, and none of the elements following it are less.

The elements are compared using `operator<` for the first version, and *comp* for the second.

Complexity

On average, linear in the distance between *first* and *last*: Compares elements, and possibly swaps (or moves) them, until the elements are properly rearranged.