

Concepts for the C++0x Standard Library: Containers

(Revision 4)

Authors: Douglas Gregor, Indiana University

Mat Marcus, Adobe Systems, Inc.

Pablo Halpern, Bloomberg, L.P.

Document number: N2776=08-0286

Revises document number: N2738=08-0248

Date: 2008-09-19

Project: Programming Language C++, Library Working Group

Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Introduction

This document proposes changes to Chapter 23 of the C++ Standard Library in order to make full use of concepts [1]. We make every attempt to provide complete backward compatibility with the pre-concept Standard Library, and note each place where we have knowingly changed semantics.

This document is formatted in the same manner as the latest working draft of the C++ standard (N2691). Future versions of this document will track the working draft and the concepts proposal as they evolve. Wherever the numbering of a (sub)section matches a section of the working paper, the text in this document should be considered replacement text, unless editorial comments state otherwise. All editorial comments will have a gray background. Changes to the replacement text are categorized and typeset as additions, removals, or changes/modifications.

Changes from N2738

- Fix definition of `push_front` in the `FrontInsertionContainer` concept map (23.1.4.3).
- Removed all uses of the term “iterator category”, in favor of explicit mentions of iterator concept names (??, 23.1.1, 23.1.2, 23.1.3, 23.2.6.1). Thanks to Alisdair Meredith for pointing out this inconsistency, and for detailed review work.
- Fixed the `AccessBack` axiom in `BackInsertionContainer` and `MemberBackInsertionContainer`. Added an additional requirement to the `*Insertion` axioms that states that the value inserted is the value now stored at the place of insertion (thanks to Alisdair Meredith).
- Convertibility requirement from `iterator` to `const_iterator` in the container concepts. Resolves concepts issue #48. Also, added convertibility from the container’s iterators reference types to its `reference` and `const_reference` types along with convertibility from the `difference_type` of a container’s iterators to its `size_type`.
- Moved `front()` operations and axioms from `BackInsertionContainer` to `Container` and provided default implementation in terms of `begin`. Moved `pop_back()` and corresponding axioms out of `BackInsertionContainer` into the (new) refined concept `StackLikeContainer`. Moved `pop_front()` and corresponding axioms

into the (new) refined concept `QueueLikeContainer()`. Similar changes were made to the Member versions of these concepts. Update constraints for `stack`, `queue`, and `priority_queue` to make use of these new concepts, avoiding queue overconstraint in the previous version of this document.

- Removed all qualification by `std::`
- Added axioms to Emplacement concepts.
- Renamed `RandomAccessAllocator` to `Allocator`
- Added `cbegin` and `cend` operations and default implementations to Container-related concepts
- The Container requirements table now states that the `X::iterator` and `X::const_iterator` types must meet the requirements of the `ForwardIterator` concept, rather than the `InputIterator` concept.
- Renamed uses of `ConstructibleAsElement` to `AllocatableElement`.
- Moved the `back()` requirement from `BackInsertionContainer` to `StackLikeContainer`.

Proposed Wording

Issues resolved by concepts

The following LWG are resolved by concepts. These issues should be resolved as NAD following the application of this proposal to the wording paper:

Issue 676. Moving the unordered containers. Applied a conceptualized version of the proposed wording. This issue was voted into the Working Paper in Bellevue, but is not in the Working Paper due to some editorial problems with the proposed wording, including some errors in the “hint” versions of the `insert` operations and the verbosity of the specification. These errors have been corrected in this document.

Issue 704. MoveAssignable requirement for container value type overly strict. The concepts proposal for the containers specifies precisely which routines require `MoveAssignable` value types.

Chapter 23 Containers library

[containers]

- 1 This clause describes components that C++ programs may use to organize collections of information.
- 2 The following subclauses describe container ~~requirements~~concepts, and components for sequences and associative containers, as summarized in Table 1:

Table 1: Containers library summary

Subclause	Header(s)
23.1 Requirements	
23.1.4 <u>Concepts</u>	<u><container_concepts></u>
23.2 Sequences	<array> <deque> <list> <queue> <stack> <vector>
23.3 Associative containers	<map> <set>
?? <code>bitset</code>	<bitset>
23.4 Unordered associative containers	<unordered_map> <unordered_set>

23.1 Container requirements

[container.requirements]

Unlike with other sections containing requirements tables, we have opted not to completely replace everything in 23.1 with a set of concepts. This decision is due to the unique nature of the container requirements, which don't really correspond to concepts because they aren't used in algorithms. Rather, the requirements tables in 23.1 are shorthand descriptions for all of the containers in this section; each container meets some subset of the requirements stated, sometimes with differing requirements on the container's value type for the same operation. Those container concepts that are actually needed (e.g., for the container adaptors 23.2.5) are specified in the new section 23.1.4. We have, however, removed many informal "requires" clauses from the container requirements tables, because they have never been complete and the same information is available more formally in the real requires clauses of the containers themselves (which appropriately account for the variation between containers).

- 3 Objects stored in these components shall be constructed using `construct_element` (??) and destroyed using the `destroy` member function of the container's allocator (??). A container may directly call constructors and destruc-

tors for its stored objects, without calling the `construct_element` or `destroy` functions, if the allocator models the `MinimalAllocator` concept. ~~For each operation that inserts an element of type T into a container (`insert`, `push_back`, `push_front`, `emplace`, etc.) with arguments `args...`, T shall be `ConstructibleAsElement`, as described in table ??.~~ [*Note:* If the component is instantiated with a scoped allocator of type A (i.e., an allocator ~~for which `is_scoped_allocator<A>::value` is true~~ that meets the requirements of the `ScopedAllocator` concept), then `construct_element` may pass an inner allocator argument to T's constructor. — end note]

- 4 ~~In table ??, T denotes an object type, A denotes an allocator, I denotes an allocator of type `A::inner_allocator_type` (if any), and `Args` denotes a template parameter pack~~

Remove Table 89: `ConstructibleAsElement<A, T, Args>` requirements

- 5 In Tables 90 and ??, X denotes a container class containing objects of type T, a and b denote values of type X, u denotes an identifier, r denotes an lvalue or a const rvalue of type X, and rv denotes a non-const rvalue of type X.

Table 90: Container requirements

expression	return type	operational semantics	assertion/note pre/post-condition	complexity
<code>X::value_type</code>	T			compile time
<code>X::reference</code>	lvalue of T			compile time
<code>X::const_reference</code>	const lvalue of T			compile time
<code>X::iterator</code>	iterator type whose value type is T		~~any iterator category except output iterator~~ meets the requirements of the <code>ForwardIterator</code> concept. convertible to <code>X::const_iterator</code> .	compile time
<code>X::const_iterator</code>	constant iterator type whose value type is T		~~any iterator category except output iterator~~ meets the requirements of the <code>ForwardIterator</code> concept	compile time
<code>X::difference_type</code>	signed integral type		is identical to the difference type of <code>X::iterator</code> and <code>X::const_iterator</code>	compile time
<code>X::size_type</code>	unsigned integral type		<code>size_type</code> can represent any non-negative value of <code>difference_type</code>	compile time
<code>X u;</code>			post: <code>u.size() == 0</code>	constant
<code>X();</code>			<code>X().size() == 0</code>	constant

expression	return type	operational semantics	assertion/note pre/post-condition	complexity
<code>X(a);</code>			~~Requires: T is CopyConstructible.~~ post: <code>a == X(a)</code> .	linear
<code>X u(a);</code> <code>X u = a;</code>			~~Requires: T is CopyConstructible.~~ post: <code>u == a</code>	linear
<code>X u(rv);</code> <code>X u = rv;</code>			~~Requires: T is MoveConstructible.~~ post: <code>u</code> shall be equal to the value that <code>rv</code> had before this construction	(Note B)
<code>a = rv;</code>	<code>X&</code>	All existing elements of <code>a</code> are either move assigned or destroyed	<code>a</code> shall be equal to the value that <code>rv</code> had before this construction	(Note C)
<code>(&a)->~X();</code>	<code>void</code>		note: the destructor is applied to every element of <code>a</code> ; all the memory is deallocated.	linear
<code>a.begin();</code>	iterator; const-iterator for constant <code>a</code>			constant
<code>a.end();</code>	iterator; const-iterator for constant <code>a</code>			constant
<code>a.cbegin();</code>	const-iterator	<code>const_cast<X const&>(a).begin();</code>		constant
<code>a.cend();</code>	const-iterator	<code>const_cast<X const&>(a).end();</code>		constant
<code>a == b</code>	convertible to <code>bool</code>		<code>==</code> is an equivalence relation. <code>a.size() == b.size() && equal(a.begin(), a.end(), b.begin())</code>	linear
<code>a != b</code>	convertible to <code>bool</code>		Equivalent to: <code>!(a == b)</code>	linear
<code>a.swap(b);</code>	<code>void</code>		<code>swap(a,b)</code>	(Note A)
<code>r = a</code>	<code>X&</code>		post: <code>r == a</code> .	linear
<code>a.size()</code>	<code>size_type</code>	<code>a.end() - a.begin()</code>		(Note A)

expression	return type	operational semantics	assertion/note pre/post-condition	complexity
a.max_size()	size_type	size() of the largest possible container		(Note A)
a.empty()	convertible to bool	a.size() == 0		constant
a < b	convertible to bool	lexicographical_compare(a.begin(), a.end(), b.begin(), b.end())	pre: < is defined for values of T. < is a total ordering relationship.	linear
a > b	convertible to bool	b < a		linear
a <= b	convertible to bool	!(a > b)		linear
a >= b	convertible to bool	!(a < b)		linear

- 11 If the iterator type of a container ~~belongs to the bidirectional or random access iterator categories (24.1)~~ meets the requirements of the [BidirectionalIterator concept](#), the container is called reversible and satisfies the additional requirements in Table 91.

23.1.1 Sequence containers

[sequence.reqmts]

- 4 The complexities of the expressions are sequence dependent.

Table 93: Sequence container requirements (in addition to container)

expression	return type	assertion/note pre/post-condition
X(n, t) X a(n, t)		~~Requires: T shall be CopyConstructible.~~ post: size() == n Constructs a sequence container with n copies of t
X(i, j) X a(i, j)		~~Requires: If the iterator's dereference operation returns an lvalue or a const rvalue, T shall be CopyConstructible.~~ Each iterator in the range [i, j) shall be dereferenced exactly once. post: size() == distance between i and j Constructs a sequence container equal to the range [i, j)
X(il)		Equivalent to X(il.begin(), il.end())
a = il	X&	a = il; return *this;

expression	return type	assertion/note pre/post-condition
<code>a.emplace(p, args)</code>	iterator	~~Requires: ConstructibleAsElement<A, T, Args>.~~ Inserts an object of type T constructed with <code>std::forward<Args>(args)...</code>
<code>a.insert(p,t)</code>	iterator	~~Requires: ConstructibleAsElement<A, T, T> and T shall be CopyAssignable.~~ Inserts a copy of t before p.
<code>a.insert(p,rv)</code>	iterator	~~Requires: ConstructibleAsElement<A, T, T&&> and T shall be MoveAssignable.~~ Inserts a copy of rv before p.
<code>a.insert(p,n,t)</code>	void	~~Requires: T shall be CopyConstructible and CopyAssignable.~~ Inserts n copies of t before p.
<code>a.insert(p,i,j)</code>	void	~~Requires: If the iterator's dereference operation returns an lvalue or a const rvalue, T shall be CopyConstructible.~~ Each iterator in the range [i, j) shall be dereferenced exactly once. pre: i and j are not iterators into a. Inserts copies of elements in [i, j) before p
<code>a.insert(p, il)</code>	void	<code>a.insert(p, il.begin(), il.end())</code>
<code>a.erase(q)</code>	iterator	~~Requires: T shall be MoveAssignable.~~ Erases the element pointed to by q
<code>a.erase(q1,q2)</code>	iterator	~~Requires: T shall be MoveAssignable.~~ Erases the elements in the range [q1, q2).
<code>a.clear()</code>	void	<code>erase(begin(), end())</code> post: <code>size() == 0</code>
<code>a.assign(i,j)</code>	void	~~Requires: If the iterator's dereference operation returns an lvalue or a const rvalue, T shall be CopyConstructible and CopyAssignable.~~ Each iterator in the range [i, j) shall be dereferenced exactly once. pre: i, j are not iterators into a. Replaces elements in a with a copy of [i, j).
<code>a.assign(il)</code>	void	<code>a.assign(il.begin(), il.end())</code>
<code>a.assign(n,t)</code>	void	~~Requires: T shall be CopyConstructible and CopyAssignable.~~ pre: t is not a reference into a. Replaces elements in a with n copies of t.

- 5 iterator and `const_iterator` types for sequence containers shall ~~be at least of the forward iterator category~~ meet the requirements of the `ForwardIterator` concept.
- 12 Table 94 lists operations that are provided for some types of sequence containers but not others. An implementation shall provide these operations for all container types shown in the “container” column, and shall implement them so as

to take amortized constant time.

Table 94: Optional sequence container operations

expression	return type	assertion/note pre/post-condition	container
<code>a.front()</code>	reference; <code>const_reference</code> for constant <code>a</code>	<code>*a.begin()</code>	vector, list, deque, basic_string
<code>a.back()</code>	reference; <code>const_reference</code> for constant <code>a</code>	{ iterator <code>tmp = a.end()</code> ; -- <code>tmp</code> ; return <code>*tmp</code> ; }	vector, list, deque, basic_string
<code>a.emplace_-</code> <code>front(args)</code>	void	<code>a.emplace(a.begin(),</code> <code>std::forward<Args>(args)...))</code> <i>Requires:</i> <i>ConstructibleAsElement<A,</i> <i>T, Args></i>	list, deque
<code>a.emplace_-</code> <code>back(args)</code>	void	<code>a.emplace(a.end(),</code> <code>std::forward<Args>(args)...))</code> <i>Requires:</i> <i>ConstructibleAsElement<A,</i> <i>T, Args></i>	list, deque, vector
<code>a.push_-</code> <code>front(t)</code>	void	<code>a.insert(a.begin(), t)</code> <i>Requires:</i> <i>ConstructibleAsElement<A,</i> <i>T, T> and T shall be</i> <i>CopyAssignable.</i>	list, deque
<code>a.push_-</code> <code>front(rv)</code>	void	<code>a.insert(a.begin(), rv)</code> <i>Requires:</i> <i>ConstructibleAsElement<A,</i> <i>T, T&&> and T shall be</i> <i>MoveAssignable.</i>	list, deque
<code>a.push_-</code> <code>back(t)</code>	void	<code>a.insert(a.end(), t)</code> <i>Requires:</i> <i>ConstructibleAsElement<A,</i> <i>T, T> and T shall be</i> <i>CopyAssignable.</i>	vector, list, deque, basic_string
<code>a.push_-</code> <code>back(rv)</code>	void	<code>a.insert(a.end(), rv)</code> <i>Requires:</i> <i>ConstructibleAsElement<A,</i> <i>T, T&&> and T shall be</i> <i>MoveAssignable.</i>	vector, list, deque, basic_string
<code>a.pop_-</code> <code>front()</code>	void	<code>a.erase(a.begin())</code>	list, deque

expression	return type	assertion/note pre/post-condition	container
a.pop_back()	void	{ iterator tmp = a.end(); --tmp; a.erase(tmp); }	vector, list, deque, basic_string
a[n]	reference; const_reference for constant a	*(a.begin() + n)	vector, deque, basic_string, match_ results
a.at(n)	reference; const_reference for constant a	*(a.begin() + n)	vector, deque

23.1.2 Associative containers

[associative.reqmts]

- 6 iterator of an associative container ~~is of the bidirectional iterator category~~ [meets the requirements of the BidirectionalIterator concept](#). For associative containers where the value type is the same as the key type, both iterator and const_iterator are constant iterators. It is unspecified whether or not iterator and const_iterator are the same type.
- 7 In Table 95, X denotes an associative container class, a denotes a value of X, a_uniq denotes a value of X when X supports unique keys, a_eq denotes a value of X when X supports multiple keys, u denotes an identifier, r denotes an lvalue or a const rvalue of type X, rv denotes a non-const rvalue of type X, i and j satisfy input iterator requirements and refer to elements implicitly convertible to value_type, [i, j) denotes a valid range, p denotes a valid const iterator to a, q denotes a valid dereferenceable const iterator to a, [q1, q2) denotes a valid range of const iterators in a, t denotes a value of X::value_type, k denotes a value of X::key_type and c denotes a value of type X::key_compare. A denotes the storage allocator used by X, if any, or std::allocator<X::value_type> otherwise, and m denotes an allocator of a type convertible to A.

Table 95: Associative container requirements (in addition to container)

expression	return type	assertion/note pre/post-condition	complexity
X::key_type	Key	~~Key is CopyConstructible and CopyAssignable~~	compile time
X::key_compare	Compare	defaults to less<key_type>	compile time
X::value_compare	a binary predicate type	is the same as key_compare for set and multiset; is an ordering relation on pairs induced by the first component (<i>i.e.</i> Key) for map and multimap.	compile time
X(c) X a(c);		~~Requires:~~ ~~ConstructibleAsElement<A, key_compare, key_compare>.~~ Constructs an empty container. Uses a copy of c as a comparison object.	constant

expression	return type	assertion/note pre/post-condition	complexity
X() X a;		<i>Requires:</i> ~~ConstructibleAsElement~~ <A, ~~key_compare~~ , ~~key_compare~~ >. Constructs an empty container. Uses Compare() as a comparison object	constant
X(i, j, c) X a(i, j, c);		<i>Requires:</i> ~~ConstructibleAsElement~~ <A, ~~key_compare~~ , ~~key_compare~~ >. Constructs an empty container and inserts elements from the range [i, j) into it; uses c as a comparison object.	$N \log N$ in general (N is the distance from i to j); linear if [i, j) is sorted with value_comp()
X(i, j) X a(i, j);		<i>Requires:</i> ~~ConstructibleAsElement~~ <A, ~~key_compare~~ , ~~key_compare~~ >. Same as above, but uses Compare() as a comparison object	same as above

No additional changes to this table.

23.1.3 Unordered associative containers

[unord.req]

- 9 In table 96: X is an unordered associative container class, a is an object of type X, b is a possibly const object of type X, a_uniq is an object of type X when X supports unique keys, a_eq is an object of type X when X supports equivalent keys, i and j are input iterators that refer to value_type, [i, j) is a valid range, p and q2 are valid const iterators to a, q and q1 are valid dereferenceable const iterators to a, [q1, q2) is a valid range in a, t is a value of type X::value_type, k is a value of type key_type, hf is a possibly const value of type hasher, eq is a possibly const value of type key_equal, n is a value of type size_type, and z is a value of type float.

Table 96: Unordered associative container requirements (in addition to container)

expression	return type	assertion/note pre/post-condition	complexity
X::key_type	Key	~~Key shall be CopyAssignable and CopyConstructible~~	compile time
X::hasher	Hash	~~Hash shall be a unary function object type such that the expression hf(k) has type std::size_t.~~	compile time
X::key_equal	Pred	~~Pred shall be a binary predicate that takes two arguments of type Key. Pred is an equivalence relation.~~	compile time

expression	return type	assertion/note	pre/post-condition	complexity
No additional changes to this table.				

- 11 The iterator types `iterator` and `const_iterator` of an unordered associative container ~~are of at least the forward iterator category~~ meet the requirements of the `ForwardIterator` concept. For unordered associative containers where the key type and value type are the same, both `iterator` and `const_iterator` are constant iterators.

Add the following new section [container.concepts]

23.1.4 Container concepts

[container.concepts]

- 1 The `container_concepts` header describes requirements on the template arguments used in container adaptors. It contains two sets of container concepts, one that uses non-member functions (23.1.4.1) and the other that uses member functions (23.1.4.2). A set of concept map templates (23.1.4.3) adapts the member-function syntax (the way most containers are implemented) to free-function syntax (which is used by most generic functions, because of its flexibility).

Header <container_concepts> synopsis

```
namespace std {
    // 23.1.4.1, container concepts
    concept Container<typename C> see below
    concept FrontInsertionContainer<typename C> see below
    concept BackInsertionContainer<typename C> see below
    concept StackLikeContainer<typename C> see below
    concept QueueLikeContainer<typename C> see below
    concept InsertionContainer<typename C> see below
    concept RangeInsertionContainer<typename C, typename Iter> see below
    concept FrontEmplacementContainer<typename C, typename... Args> see below
    concept BackEmplacementContainer<typename C, typename... Args> see below
    concept EmplacementContainer<typename C, typename... Args> see below

    // 23.1.4.2, member container concepts
    auto concept MemberContainer<typename C> see below
    auto concept MemberFrontInsertionContainer<typename C> see below
    auto concept MemberBackInsertionContainer<typename C> see below
    auto concept MemberStackLikeContainer<typename C> see below
    auto concept MemberQueueLikeContainer<typename C> see below
    auto concept MemberInsertionContainer<typename C> see below
    auto concept MemberRangeInsertionContainer<typename C, typename Iter> see below
    auto concept MemberFrontEmplacementContainer<typename C, typename... Args> see below
    auto concept MemberBackEmplacementContainer<typename C, typename... Args> see below
    auto concept MemberEmplacementContainer<typename C, typename... Args> see below

    // 23.1.4.3, container concept maps
    template <MemberContainer C> concept_map Container<C> see below
    template <MemberFrontInsertionContainer C> concept_map FrontInsertionContainer<C> see below
    template <MemberBackInsertionContainer C> concept_map BackInsertionContainer<C> see below
    template <MemberStackLikeContainer C> concept_map StackLikeContainer<C> see below
}
```

```

template <MemberQueueLikeContainer C> concept_map QueueLikeContainer<C> see below
template <MemberInsertionContainer C> concept_map InsertionContainer<C> see below
template <MemberRangeInsertionContainer C, InputIterator Iter>
    concept_map RangeInsertionContainer<C, Iter> see below
template <MemberFrontEmplacementContainer C, typename... Args>
    concept_map FrontEmplacementContainer<C, Args...> see below
template <MemberBackEmplacementContainer C, typename... Args>
    concept_map BackEmplacementContainer<C, Args...> see below
template <MemberEmplacementContainer C, typename... Args>
    concept_map EmplacementContainer<C, Args...> see below
template <typename E, size_t N> concept_map Container<E[N]> see below
template <typename E, size_t N> concept_map Container<const E[N]> see below
}

```

23.1.4.1 Free function container concepts

[container.concepts.free]

- 1 This section contains the container concepts that are used by other parts of the library. These concepts are written in terms of free functions. For backward compatibility, member function versions and concept maps adapting member to free syntax follow in (23.1.4.2) and (23.1.4.3).

```

concept Container<typename C> {
    ObjectType          value_type      = typename C::value_type;
    typename           reference       = typename C::reference;
    typename           const_reference = typename C::const_reference;
    UnsignedIntegralLike size_type      = typename C::size_type;

    ForwardIterator iterator;
    ForwardIterator const_iterator;

    requires Convertible<reference, const_reference>
        && Convertible<reference, const value_type&>
        && Convertible<const_reference, const _value_type&>;
        && Convertible<iterator, const_iterator>
        && SameType<ForwardIterator<iterator>::value_type, value_type>
        && SameType<ForwardIterator<const_iterator>::value_type, value_type>
        && Convertible<ForwardIterator<iterator>::reference, reference>
        && Convertible<ForwardIterator<const_iterator>::reference, const_reference>
        && SameType<ForwardIterator<iterator>::difference_type,
            ForwardIterator<const_iterator>::difference_type>
        && IntegralType<size_type>
        && Convertible<ForwardIterator<iterator>::difference_type, size_type>;

    bool          empty(const C& c) { return begin(c) == end(c); }
    size_type     size(const C& c)  { return distance(begin(c), end(c)); }

    iterator      begin(C&);
    const_iterator begin(const C&);
    iterator      end(C&);
    const_iterator end(const C&);
    const_iterator cbegin(const C& c) { return begin(c); }
}

```

```

const_iterator cend(const C& c) { return end(c); }
reference      front(C& c) { return *begin(c); }
const_reference front(const C& c) { return *begin(c); }

```

```

axiom AccessFront(C c) {
    if (begin(c) != end(c)) front(c) == *begin(c);
}

```

```

axiom ContainerSize(C c) {
    (begin(c) == end(c)) == empty(c);
    (begin(c) != end(c)) == (size(c) > 0);
}

```

2 *Note:* describes a container, which provides iteration through a sequence of elements stored in the container.

3 *Requires:* for a (possibly const-qualified) container *c*, *[begin(c), end(c))* is a valid range.

```

concept FrontInsertionContainer<typename C> : Container<C> {
    void push_front(C&, value_type&&);

    axiom FrontInsertion(C c, value_type x) {
        x == (push_front(c, x), front(c));
    }
}

```

4 *Note:* describes a container that can be modified by adding elements to the front of the sequence.

```

concept BackInsertionContainer<typename C> : Container<C> {
    void push_back(C&, value_type&&);
}

```

5 *Note:* describes a container that can be modified by adding to the back of the sequence.

```

concept StackLikeContainer<typename C> : BackInsertionContainer<C> {
    reference      back(C&);
    const_reference back(const C&);

    void pop_back(C&);

    requires BidirectionalIterator<iterator> axiom AccessBack(C c) {
        if (begin(c) != end(c)) back(c) == * (--end(c));
    }

    axiom BackInsertion(C c, value_type x) {
        x == (push_back(c, x), back(c));
    }

    axiom BackRemoval(C c, value_type x) {
        c == (push_back(c, x), pop_back(c), c);
    }
}

```

}

- 6 *Note:* describes a container that can be modified by adding or removing elements from the back of the sequence.

```
concept QueueLikeContainer<typename C> : BackInsertionContainer<C> {
    void pop_front(C&);
}
```

- 7 *Note:* describes a container that can be modified by adding elements to the back or removing elements from the front of the sequence.

```
concept InsertionContainer<typename C> : Container<C> {
    iterator insert(C&, const_iterator, value_type&&);

    axiom Insertion(C c, const_iterator position, value_type v) {
        v == *insert(c, position, v);
    }
}
```

- 8 *Note:* describes a container that can be modified by inserting elements at any position within the sequence.

```
concept RangeInsertionContainer<typename C, typename Iter> : InsertionContainer<C> {
    requires InputIterator<Iter>;
    void insert(C&, const_iterator position, Iter first, Iter last);
}
```

- 9 *Note:* describes a container that can be modified by inserting a sequence of elements at any position within the sequence.

```
concept FrontEmplacementContainer<typename C, typename... Args> : Container<C> {
    void emplace_front(C& c, Args&&... args);

    requires Constructible<value_type, Args...>
    axiom FrontEmplacement(C c, Args... args) {
        value_type(args...) == (emplace_front(c, args...), front(c));
    }

    requires FrontInsertionContainer<C> && Constructible<value_type, Args...>
    axiom FrontEmplacementPushEquivalence(C c, Args... args) {
        (emplace_front(c, args...), front(c)) == (push_front(c, value_type(args...)), front(c));
    }
}
```

- 10 *Note:* describes a container that can be modified by constructing elements at the front of the sequence.

```
concept BackEmplacementContainer<typename C, typename... Args> : Container<C> {
    void emplace_back(C& c, Args&&... args);

    requires StackLikeContainer<C> && Constructible<value_type, Args...>
    axiom BackEmplacement(C c, Args... args) {
        value_type(args...) == (emplace_back(c, args...), back(c));
    }
}
```

```

requires StackLikeContainer<C> && Constructible<value_type, Args...>
    axiom BackEmplacementPushEquivalence(C c, Args... args) {
        (emplace_back(c, args...), back(c)) == (push_back(c, value_type(args...)), back(c));
    }
}

```

11 *Note:* describes a container that can be modified by constructing elements at the back of the sequence.

```

concept EmplacementContainer<typename C, typename... Args> : Container<C> {
    iterator emplace(C& c, const_iterator position, Args&&... args);

    requires Constructible<value_type, Args...>
        axiom Emplacement(C c, const_iterator position, Args... args) {
            value_type(args...) == *emplace(c, position, args...);
        }

    requires InsertionContainer<C> && Constructible<value_type, Args...>
        axiom EmplacementPushEquivalence(C c, const_iterator position, Args... args) {
            *emplace(c, position, args...) == *insert(c, position, value_type(args...));
        }
}

```

12 *Note:* describes a container that can be modified by constructing elements at any position within the sequence.

23.1.4.2 Member container concepts

[container.concepts.member]

- 1 This section contains backward compatibility concepts, written using member function syntax, corresponding to the container concepts (23.1.4.1). Concept maps that automatically adapt these member function concepts to the free function concept syntax follow (23.1.4.3).

```

auto concept MemberContainer<typename C> {
    ObjectType          value_type      = typename C::value_type;
    typename            reference       = typename C::reference;
    typename            const_reference = typename C::const_reference;
    UnsignedIntegralLike size_type      = typename C::size_type;

    ForwardIterator iterator;
    ForwardIterator const_iterator;

    requires Convertible<reference, const_reference>
        && Convertible<reference, const value_type&>
        && Convertible<const_reference, const value_type&>;
        && Convertible<iterator, const_iterator>
        && SameType<ForwardIterator<iterator>::value_type, value_type>
        && SameType<ForwardIterator<const_iterator>::value_type, value_type>
        && Convertible<ForwardIterator<iterator>::reference, reference>
        && Convertible<ForwardIterator<const_iterator>::reference, const_reference>
        && SameType<ForwardIterator<iterator>::difference_type,
            ForwardIterator<const_iterator>::difference_type>
}

```

```

    && IntegralType<size_type>
    && Convertible<ForwardIterator<iterator>::difference_type, size_type>;

    bool          C::empty() const { return  this->begin() == this->end(); }
    size_type      C::size() const  { return  distance(this->begin(), this->end()); }

    iterator       C::begin();
    const_iterator C::begin() const;
    iterator       C::end();
    const_iterator C::end() const;
    const_iterator C::cbegin() const { return this->begin(); }
    const_iterator C::cend() const   { return this->end(); }
    reference      C::front()        { return *this->begin(); }
    const_reference C::front() const  { return *this->begin(); }

    axiom MemberAccessFront(C c) {
        if (c.begin() != c.end()) c.front() == *c.begin();
    }

    axiom MemberContainerSize(C c) {
        (c.begin() == c.end()) == c.empty();
        (c.begin() != c.end()) == (c.size() > 0);
    }
}

```

- 2 *Note:* describes a container, in terms of member functions, which provides iteration through a sequence of elements stored in the container. *Requires:* for a (possibly const-qualified) container *c*, [*c.begin()*, *c.end()*) is a valid range.

```

auto concept MemberFrontInsertionContainer<typename C> : MemberContainer<C> {
    void C::push_front(value_type&&);

    axiom MemberFrontInsertion(C c, value_type x) {
        x == (c.push_front(x), c.front());
    }
}

```

- 4 *Note:* describes a container, in terms of member functions, that can be modified by adding elements to the front of the container.

```

auto concept MemberBackInsertionContainer<typename C> : MemberContainer<C> {
    void C::push_back(value_type&&);
}

```

- 5 *Note:* describes a container, in terms of member functions, that can be modified by adding elements to the back of the container.

```

auto concept MemberStackLikeContainer<typename C> : MemberBackInsertionContainer<C> {
    reference      C::back();
    const_reference C::back() const;
}

```

```

void C::pop_back();

requires BidirectionalIterator<iterator> axiom MemberAccessBack(C c) {
    if (c.begin() != c.end()) c.back() == *(--c.end());
}

axiom MemberBackInsertion(C c, value_type x) {
    x == (c.push_back(x), c.back());
}

axiom MemberBackRemoval(C c, value_type x) {
    c == (c.push_back(x), c.pop_back(), c);
}
}

```

- 6 *Note:* describes a container, in terms of member functions, that can be modified by adding or removing elements from the back of the container.

```

auto concept MemberQueueLikeContainer<typename C> : MemberBackInsertionContainer<C> {
    void C::pop_front();
}

```

- 7 *Note:* describes a container, in terms of member functions, that can be modified by adding elements to the back or removing elements from the front of the container.

```

auto concept MemberInsertionContainer<typename C> : MemberContainer<C> {
    iterator C::insert(const_iterator, value_type&&);

    axiom MemberInsertion(C c, const_iterator position, value_type v) {
        v == *c.insert(position, v);
    }
}

```

- 8 *Note:* describes a container, in terms of member functions, that can be modified by inserting elements at any position within the container.

```

auto concept MemberRangeInsertionContainer<typename C, typename Iter> : MemberInsertionContainer<C> {
    requires InputIterator<Iter>;
    void C::insert(const_iterator position, Iter first, Iter last);
}

```

- 9 *Note:* describes a container, in terms of member functions, that can be modified by inserting a sequence of elements at any position within the sequence.

```

auto concept MemberFrontEmplacementContainer<typename C, typename... Args> : MemberContainer<C> {
    void C::emplace_front(Args&&... args);

    requires Constructible<value_type, Args...>
    axiom MemberFrontEmplacement(C c, Args... args) {
        value_type(args...) == (c.emplace_front(args...), c.front());
    }
}

```

```

requires MemberFrontInsertionContainer<C> && Constructible<value_type, Args...>
axiom MemberFrontEmplacementPushEquivalence(C c, Args... args) {
    (c.emplace_front(args...), c.front()) == (c.push_front(value_type(args...)), c.front());
}
}

```

- 10 *Note:* describes a container, in terms of member functions, that can be modified by placing a newly-constructed object at the front of the sequence.

```

auto concept MemberBackEmplacementContainer<typename C, typename... Args> : MemberBackInsertionContainer<C> {
    void C::emplace_back(Args&&... args);

    requires MemberStackLikeContainer<C> && Constructible<value_type, Args...>
    axiom MemberBackEmplacement(C c, Args... args) {
        value_type(args...) == (c.emplace_back(args...), c.back());
    }

    requires MemberStackLikeContainer<C> && Constructible<value_type, Args...>
    axiom MemberBackEmplacementPushEquivalence(C c, Args... args) {
        (c.emplace_back(args...), c.back()) == (c.push_back(value_type(args...)), c.back());
    }
}

```

- 11 *Note:* describes a container, in terms of member functions, that can be modified by constructing elements at the back of the sequence.

```

auto concept MemberEmplacementContainer<typename C, typename... Args> : MemberInsertionContainer<C> {
    void C::emplace(const_iterator position, Args&&... args);

    require Constructible<value_type, Args...>
    axiom MemberEmplacement(C c, const_iterator position, Args... args) {
        value_type(args...) == *c.emplace(position, args...);
    }

    requires MemberInsertionContainer<C> && Constructible<value_type, Args...>
    axiom MemberEmplacementPushEquivalence(C c, const_iterator position, Args... args) {
        *c.emplace(position, args...) == *c.insert(position, value_type(args...));
    }
}

```

- 12 *Note:* describes a container, in terms of member functions, that can be modified by constructing elements at any position within the sequence.

23.1.4.3 Container concept maps

[container.concepts.maps]

- 1 This section contains concept maps that automatically adapt classes with the appropriate member functions, as specified in (23.1.4.2), to meet the free function container concept syntax in (23.1.4.1). It also contains maps adapting built-in arrays to model the appropriate container concepts, and maps adapting emplacement container concepts to model insertion container concepts.

```

template <MemberContainer C>
concept_map Container<C> {
    typedef C::value_type      value_type;
    typedef C::reference       reference;
    typedef C::const_reference const_reference;
    typedef C::size_type       size_type;

    typedef C::iterator        iterator;
    typedef C::const_iterator  const_iterator;

    bool      empty(const C& c)  { return c.empty(); }
    size_type size(const C& c)  { return c.size(); }

    iterator  begin(C& c)        { return c.begin(); }
    const_iterator begin(const C& c) { return c.begin(); }
    iterator  end(C& c)          { return c.end(); }
    const_iterator end(const C& c) { return c.end(); }
    const_iterator cbegin(const C& c) { return c.cbegin(); }
    const_iterator cend(const C& c) { return c.cend(); }
    reference  front(C& c)       { return c.front(); }
    const_reference front(const C& c) { return c.front(); }
}

```

- 2 *Note:* Adapts an existing container, which uses member function syntax for each of its operations, to the Container concept.

```

template <MemberFrontInsertionContainer C>
concept_map FrontInsertionContainer<C> {
    typedef Container<C>::value_type value_type;

    void push_front(C& c, value_type&& v) { c.push_front(static_cast<value_type&&>(v)); }
}

```

- 3 *Note:* Adapts an existing container, which uses member function syntax for each of its operations, to the FrontInsertionContainer concept.

```

template <MemberBackInsertionContainer C>
concept_map BackInsertionContainer<C> {
    typedef Container<C>::value_type value_type;

    void push_back(C& c, value_type&& v) { c.push_back(static_cast<value_type&&>(v)); }
}

```

- 4 *Note:* Adapts an existing container, which uses member function syntax for each of its operations, to the BackInsertionContainer concept.

```

template <MemberStackLikeContainer C>
concept_map StackLikeContainer<C> {
    typedef Container<C>::reference      reference;
    typedef Container<C>::const_reference const_reference;
}

```

```

reference      back(C& c)      { return c.back(); }
const_reference back(const C& c) { return c.back(); }
void          pop_back(C& c)   { c.pop_back(); }
}

```

- 5 *Note:* Adapts an existing container, which uses member function syntax for each of its operations, to the `StackLikeContainer` concept.

```

template <MemberQueueLikeContainer C>
concept_map QueueLikeContainer<C> {
    void pop_front(C& c) { c.pop_front(); }
}

```

- 6 *Note:* Adapts an existing container, which uses member function syntax for each of its operations, to the `QueueLikeContainer` concept.

```

template <MemberInsertionContainer C>
concept_map InsertionContainer<C> {
    typedef Container<C>::value_type value_type;
    Container<C>::iterator insert(C& c, Container<C>::const_iterator i, value_type&& v)
    { return c.insert(i, static_cast<value_type&&>(v)); }
}

```

- 7 *Note:* Adapts an existing insertion container, which uses member function syntax for each of its operations, to the `InsertionContainer` concept.

```

template <MemberRangeInsertionContainer C, InputIterator Iter>
concept_map RangeInsertionContainer<C, Iter> {
    void insert(C& c, Container<C>::const_iterator i, Iter first, Iter last)
    { c.insert(i, first, last); }
}

```

- 8 *Note:* Adapts an existing range-insertion container, which uses member function syntax for each of its operations, to the `RangeInsertionContainer` concept.

```

template <MemberFrontEmplacementContainer C, typename... Args>
concept_map FrontEmplacementContainer<C, Args...> {
    void emplace_front(C& c, Args&&... args)
    { c.emplace_front(forward<Args>(args)...); }
}

```

- 9 *Note:* Adapts an existing front-emplace container, which uses member function syntax for each of its operations, to the `FrontEmplacementContainer` concept.

```

template <MemberBackEmplacementContainer C, typename... Args>
concept_map BackEmplacementContainer<C, Args...> {
    void emplace_back(C& c, Args&&... args)
    { c.emplace_back(forward<Args>(args)...); }
}

```

- 10 *Note:* Adapts an existing back-emplace container, which uses member function syntax for each of its operations, to the `BackEmplacementContainer` concept.

```
template <MemberEmplacementContainer C, typename... Args>
concept_map EmplacementContainer<C, Args...> {
    Container<C>::iterator emplace(C& c, Container<C>::const_iterator position, Args&&... args)
    { return c.emplace(position, forward<Args>(args)...); }
}
```

- 11 *Note:* Adapts an existing `emplace` container, which uses member function syntax for each of its operations, to the `EmplacementContainer` concept.

```
template <typename E, size_t N>
concept_map Container<E[N]> {
    typedef E                value_type;
    typedef E&               reference;
    typedef const E&         const_reference;
    typedef size_t           size_type;
    typedef E*               iterator;
    typedef const E*         const_iterator;

    bool                empty(const E(&c)[N]) { return N==0; }
    size_type           size(const E(&c)[N]) { return N; }

    iterator            begin(E(&c)[N])        { return c; }
    const_iterator      begin(const E(&c)[N])    { return c; }
    iterator            end(E(&c)[N])            { return c + N; }
    const_iterator      end(const E(&c)[N])      { return c + N; }
}
```

```
template <typename E, size_t N>
concept_map Container<const E[N]> {
    typedef E                value_type;
    typedef const E&         reference;
    typedef const E&         const_reference;
    typedef size_t           size_type;

    typedef const E*         iterator;
    typedef const E*         const_iterator;

    bool                empty(const E(&c)[N]) { return N==0; }
    size_type           size(const E(&c)[N]) { return N; }

    const_iterator      begin(const E(&c)[N])    { return c; }
    const_iterator      end(const E(&c)[N])      { return c + N; }
}
```

- 12 *Note:* Adapts built-in arrays to the `Container` concept.

23.2 Sequences

[sequences]

- 1 Headers `<array>`, `<deque>`, `<forward_list>`, `<list>`, `<queue>`, `<stack>`, and `<vector>`.

Header <array> synopsis

```

namespace std {
    template <class ValueType T, size_t N >
        requires NothrowDestructible<T>
        struct array;
    template <class EqualityComparable T, size_t N>
        bool operator==(const array<T,N>& x, const array<T,N>& y);
    template <class EqualityComparable T, size_t N>
        bool operator!=(const array<T,N>& x, const array<T,N>& y);
    template <class LessThanComparable T, size_t N>
        bool operator<(const array<T,N>& x, const array<T,N>& y);
    template <class LessThanComparable T, size_t N>
        bool operator>(const array<T,N>& x, const array<T,N>& y);
    template <class LessThanComparable T, size_t N>
        bool operator<=(const array<T,N>& x, const array<T,N>& y);
    template <class LessThanComparable T, size_t N>
        bool operator>=(const array<T,N>& x, const array<T,N>& y);
    template <class Swappable T, size_t N >
        void swap(array<T,N>& x, array<T,N>& y);

    template <class ObjectType T> class tuple_size;
    template <size_t I, class ObjectType T>
        class tuple_element;
    template <class ObjectType T, size_t N>
        struct tuple_size<array<T, N> >;
    template <size_t I, class T, size_t N>
        requires True<(I < N)>
        struct tuple_element<I, array<T, N> >;
    template <size_t I, class T, size_t N>
        requires True<(I < N)>
        T& get(array<T, N>&);
    template <size_t I, class T, size_t N>
        requires True<(I < N)>
        const T& get(const array<T, N>&);
}

```

Header <deque> synopsis

```

namespace std {
    template <class ValueType T, class Allocator Allocator = allocator<T> >
        requires NothrowDestructible<T>
        class deque;
    template <class EqualityComparable T, class Allocator>
        bool operator==(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
    template <class LessThanComparable T, class Allocator>
        bool operator<(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
    template <class EqualityComparable T, class Allocator>
        bool operator!=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
    template <class LessThanComparable T, class Allocator>
        bool operator>(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
}

```

```

template <classLessThanComparable T, class Allocator>
    bool operator>=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
template <classLessThanComparable T, class Allocator>
    bool operator<=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
template <classObjectType T, class Allocator>
    void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
template <classObjectType T, class Allocator>
    void swap(deque<T,Allocator>&& x, deque<T,Allocator>& y);
template <classObjectType T, class Allocator>
    void swap(deque<T,Allocator>& x, deque<T,Allocator>&& y);
}

```

Header <forward_list> synopsis

```

namespace std {
    template <classValueType T, classAllocator Allocator = allocator<T> >
        requires NothrowDestructible<T>
        class forward_list;
    template <classEqualityComparable T, class Allocator>
        bool operator==(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
    template <classLessThanComparable T, class Allocator>
        bool operator< (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
    template <classEqualityComparable T, class Allocator>
        bool operator!=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
    template <classLessThanComparable T, class Allocator>
        bool operator> (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
    template <classLessThanComparable T, class Allocator>
        bool operator>=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
    template <classLessThanComparable T, class Allocator>
        bool operator<=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
    template <classValueType T, class Allocator>
        void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
}

```

Header <list> synopsis

```

namespace std {
    template <classValueType T, classAllocator Allocator = allocator<T> >
        requires NothrowDestructible<T>
        class list;
    template <classEqualityComparable T, class Allocator>
        bool operator==(const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <classLessThanComparable T, class Allocator>
        bool operator< (const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <classEqualityComparable T, class Allocator>
        bool operator!=(const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <classLessThanComparable T, class Allocator>
        bool operator> (const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <classLessThanComparable T, class Allocator>
        bool operator>=(const list<T,Allocator>& x, const list<T,Allocator>& y);
    template <classLessThanComparable T, class Allocator>
        bool operator<=(const list<T,Allocator>& x, const list<T,Allocator>& y);
}

```

```

    bool operator<=(const list<T,Allocator>& x, const list<T,Allocator>& y);
template <classValueType T, class Allocator>
    void swap(list<T,Allocator>& x, list<T,Allocator>& y);
template <classValueType T, class Allocator>
    void swap(list<T,Allocator>&& x, list<T,Allocator>& y);
template <classValueType T, class Allocator>
    void swap(list<T,Allocator>& x, list<T,Allocator>&& y);
}

```

Header <queue> synopsis

```

namespace std {
    template <classObjectType T, class Container = deque<T> >
        requires QueueLikeContainer<Cont>
            && SameType<T, Cont::value_type>
            && NothrowDestructible<Cont>
        class queue;
    template <class T, classEqualityComparable Container>
        bool operator==(const queue<T, Container>& x, const queue<T, Container>& y);
    template <class T, classLessThanComparable Container>
        bool operator< (const queue<T, Container>& x, const queue<T, Container>& y);
    template <class T, classEqualityComparable Container>
        bool operator!=(const queue<T, Container>& x, const queue<T, Container>& y);
    template <class T, classLessThanComparable Container>
        bool operator> (const queue<T, Container>& x, const queue<T, Container>& y);
    template <class T, classLessThanComparable Container>
        bool operator>= (const queue<T, Container>& x, const queue<T, Container>& y);
    template <class T, classLessThanComparable Container>
        bool operator<= (const queue<T, Container>& x, const queue<T, Container>& y);
    template <classObjectType T, classSwappable Container>
        void swap(queue<T, Container>& x, queue<T, Container>& y);
    template <classObjectType T, classSwappable Container>
        void swap(queue<T, Container>&& x, queue<T, Container>& y);
    template <classObjectType T, classSwappable Container>
        void swap(queue<T, Container>& x, queue<T, Container>&& y);

    template <classObjectType T, classStackLikeContainer Container = vector<T>,
        classStrictWeakOrder<auto, T> Compare = less<typename Container::value_type> >
        requires SameType<Cont::value_type, T> && RandomAccessIterator<Cont::iterator>
            && ShuffleIterator<Cont::iterator> && CopyConstructible<Compare>
            && NothrowDestructible<Cont>
        class priority_queue;
    template <classObjectType T, classSwappable Container, classSwappable Compare>
        void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>& y);
    template <classObjectType T, classSwappable Container, classSwappable Compare>
        void swap(priority_queue<T, Container, Compare>&& x, priority_queue<T, Container, Compare>& y);
    template <classObjectType T, classSwappable Container, classSwappable Compare>
        void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>&& y);
}

```

Header <stack> synopsis

```

namespace std {
    template <class ObjectType T, class StackLikeContainer Container = deque<T> >
        requires SameType<Cont::value_type, T>
            && NothrowDestructible<Cont>
        class stack;
    template <class T, class EqualityComparable Container>
        bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);
    template <class T, class LessThanComparable Container>
        bool operator< (const stack<T, Container>& x, const stack<T, Container>& y);
    template <class T, class EqualityComparable Container>
        bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);
    template <class T, class LessThanComparable Container>
        bool operator> (const stack<T, Container>& x, const stack<T, Container>& y);
    template <class T, class LessThanComparable Container>
        bool operator>= (const stack<T, Container>& x, const stack<T, Container>& y);
    template <class T, class LessThanComparable Container>
        bool operator<= (const stack<T, Container>& x, const stack<T, Container>& y);
    template <class ObjectType T, class Swappable Container>
        void swap(stack<T, AllocatorCont>& x, stack<T, AllocatorCont>& y);
    template <class ObjectType T, class Swappable Container>
        void swap(stack<T, Container>&& x, stack<T, Container>&& y);
    template <class ObjectType T, class Swappable Container>
        void swap(stack<T, Container>& x, stack<T, Container>&& y);
}

```

Header <vector> synopsis

```

namespace std {
    template <class ValueType T, class Allocator Allocator = allocator<T> >
        requires MoveConstructible<T>
        class vector;
    template <class EqualityComparable T, class Allocator>
        bool operator==(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
    template <class LessThanComparable T, class Allocator>
        bool operator< (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
    template <class EqualityComparable T, class Allocator>
        bool operator!=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
    template <class LessThanComparable T, class Allocator>
        bool operator> (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
    template <class LessThanComparable T, class Allocator>
        bool operator>= (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
    template <class LessThanComparable T, class Allocator>
        bool operator<= (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
    template <class ValueType T, class Allocator>
        void swap(vector<T, Allocator>& x, vector<T, Allocator>& y);
    template <class ValueType T, class Allocator>
        void swap(vector<T, Allocator>&& x, vector<T, Allocator>&& y);
    template <class ValueType T, class Allocator>
        void swap(vector<T, Allocator>& x, vector<T, Allocator>&& y);
}

```

```
template <class Allocator Allocator> class vector<bool, Allocator>;
```

23.2.1 Class template array

[array]

- 1 The header <array> defines a class template for storing fixed-size sequences of objects. An array supports random access iterators. An instance of array<T, N> stores N elements of type T, so that size() == N is an invariant. The elements of an array are stored contiguously, meaning that if a is an array<T, N> then it obeys the identity &a[n] == &a[0] + n for all 0 ≤ n < N.
- 2 An array is an aggregate (??) that can be initialized with the syntax

```
array a = { initializer-list };
```

where *initializer-list* is a comma separated list of up to N elements whose types are convertible to T.

- 3 Unless otherwise specified, all array operations are as described in 23.1. Descriptions are provided here only for operations on array that are not described in that clause or for operations where there is additional semantic information.

```
namespace std {
    template <class ValueType T, size_t N >
    requires NothrowDestructible<T>
    struct array {
        // types:
        typedef T &                                reference;
        typedef const T &                          const_reference;
        typedef implementation_defined              iterator;
        typedef implementation_defined              const_iterator;
        typedef size_t                              size_type;
        typedef ptrdiff_t                          difference_type;
        typedef T                                    value_type;
        typedef reverse_iterator<iterator>          reverse_iterator;
        typedef reverse_iterator<const_iterator>    const_reverse_iterator;

        T        elems[N];                        // exposition only

        // No explicit construct/copy/destroy for aggregate type

        requires CopyAssignable<T> void assign(const T& u);
        requires Swappable<T> void swap(array<T, N> &);

        // iterators:
        iterator        begin();
        const_iterator  begin() const;
        iterator        end();
        const_iterator  end() const;

        reverse_iterator  rbegin();
        const_reverse_iterator  rbegin() const;
        reverse_iterator  rend();
        const_reverse_iterator  rend() const;
    };
}
```

```

const_iterator      cbegin() const;
const_iterator      cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
constexpr size_type size() const;
constexpr size_type max_size() const;
bool      empty() const;

// element access:
reference      operator[] (size_type n);
const_reference operator[] (size_type n) const;
const_reference at(size_type n) const;
reference      at(size_type n);
reference      front();
const_reference front() const;
reference      back();
const_reference back() const;

T *      data();
const T * data() const;
};
}

```

- 4 [Note: The member variable `elems` is shown for exposition only, to emphasize that `array` is a class aggregate. The name `elems` is not part of `array`'s interface. — end note]

23.2.1.1 `array` constructors, copy, and assignment

[array.cons]

- 1 The conditions for an aggregate (??) shall be met. Class `array` relies on the implicitly-declared special member functions (??, ??, and ??) to conform to the container requirements table in 23.1.

23.2.1.2 `array` specialized algorithms

[array.special]

```
template <class Swappable T, size_t N> void swap(array<T,N>& x, array<T,N>& y);
```

- 1 *Effects:*

```
swap_ranges(x.begin(), x.end(), y.begin() );
```

23.2.1.3 `array::size`

[array.size]

```
template <class T, size_t N> size_type array<T,N>::size();
```

- 1 *Returns:* N

23.2.1.4 `array::data`

[array.data]

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

1 *Returns:* elems.

23.2.1.5 Zero sized arrays

[array.zero]

- 1 array shall provide support for the special case $N == 0$.
- 2 In the case that $N == 0$, `begin()` == `end()` == unique value. The return value of `data()` is unspecified.
- 3 The effect of calling `front()` or `back()` for a zero-sized array is implementation defined.

23.2.1.6 Tuple interface to class template array

[array.tuple]

```
tuple_size<array<T, N> >::value
```

1 *Return type:* integral constant expression.

2 *Value:* N

```
tuple_element<I, array<T, N> >::type
```

3 *Requires:* $0 \leq I < N$. The program is ill-formed if I is out of bounds.

4 *Value:* The type T.

```
template <size_t I, class T, size_t N>
requires True<(I < N)>
T& get(array<T, N>& a);
```

5 ~~*Requires:* $I < N$. The program is ill-formed if I is out of bounds.~~

Returns: A reference to the Ith element of a, where indexing is zero-based.

Throws: nothing.

```
template <size_t I, class T, size_t N>
requires True<(I < N)>
const T& get(const array<T, N>& a);
```

6 ~~*Requires:* $I < N$. The program is ill-formed if I is out of bounds.~~

7 *Returns:* A const reference to the Ith element of a, where indexing is zero-based.

Throws: nothing.

23.2.2 Class template deque

[deque]

- 1 A deque is a sequence container that, like a vector (23.2.6), supports random access iterators. In addition, it supports constant time insert and erase operations at the beginning or the end; insert and erase in the middle take linear time. That is, a deque is especially optimized for pushing and popping elements at the beginning and end. As with vectors, storage management is handled automatically.

- 2 A deque satisfies all of the requirements of a container, of a reversible container (given in tables in 23.1), of a sequence container, including the optional sequence container requirements (23.1.1), and of an allocator-aware container (Table ??). Descriptions are provided here only for operations on deque that are not described in one of these tables or for operations where there is additional semantic information.

```

namespace std {
    template <class ValueType T, class Allocator Allocator = allocator<T> >
    requires NothrowDestructible<T>
    class deque {
    public:
        // types:
        typedef typename Allocator::reference          reference;
        typedef typename Allocator::const_reference    const_reference;
        typedef implementation-defined                 iterator;           // See 23.1
        typedef implementation-defined                 const_iterator;     // See 23.1
        typedef implementation-defined                 size_type;           // See 23.1
        typedef implementation-defined                 difference_type;      // See 23.1
        typedef T                                       value_type;
        typedef Allocator                              allocator_type;
        typedef typename Allocator::pointer            pointer;
        typedef typename Allocator::const_pointer      const_pointer;
        typedef reverse_iterator<iterator>             reverse_iterator;
        typedef reverse_iterator<const_iterator>       const_reverse_iterator;

        // 23.2.2.1 construct/copy/destroy:
        explicit deque(const Allocator& = Allocator());
        requires AllocatableElement<Alloc, T> explicit deque(size_type n);
        requires AllocatableElement<Alloc, T, const T&>
        deque(size_type n, const T& value, const Allocator& = Allocator());
        template <class InputIterator InputIterator Iter>
        requires AllocatableElement<Alloc, T, Iter::reference>
        deque(InputIterator first, InputIterator last, const Allocator& = Allocator());
        requires AllocatableElement<Alloc, T, const T&> deque(const deque<T, Allocator>& x);
        requires AllocatableElement<Alloc, T, T&&> deque(deque&&);
        requires AllocatableElement<Alloc, T, const T&> deque(const deque&, const Allocator&);
        requires AllocatableElement<Alloc, T, T&&> deque(deque&&, const Allocator&);
        requires AllocatableElement<Alloc, T, const T&>
        deque(initializer_list<T>, const Allocator& = Allocator());

        ~deque();
        requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
        deque<T, Allocator>& operator=(const deque<T, Allocator>& x);
        requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
        deque<T, Allocator>& operator=(const deque<T, Allocator>&& x);
        requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
        deque& operator=(initializer_list<T>);
        template <class InputIterator InputIterator Iter>
        requires AllocatableElement<Alloc, T, Iter::reference>
        && HasAssign<T, Iter::reference>
        void assign(InputIterator first, InputIterator last);
    
```

```

requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
    void assign(size_type n, const T& t);
requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
    void assign(initializer_list<T>);
allocator_type get_allocator() const;

// iterators:
iterator          begin();
const_iterator    begin() const;
iterator          end();
const_iterator    end() const;
reverse_iterator  rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator  rend();
const_reverse_iterator rend() const;

const_iterator    cbegin() const;
const_iterator    cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// 23.2.2.2 capacity:
size_type size() const;
size_type max_size() const;
requires AllocatableElement<Alloc, T>
    void resize(size_type sz);
requires AllocatableElement<Alloc, T, const T&>
    void resize(size_type sz, const T& c);
bool empty() const;

// element access:
reference          operator[] (size_type n);
const_reference    operator[] (size_type n) const;
reference          at(size_type n);
const_reference    at(size_type n) const;
reference          front();
const_reference    front() const;
reference          back();
const_reference    back() const;

// 23.2.2.3 modifiers:
template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    void emplace_front(Args&&... args);
template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    void emplace_back(Args&&... args);

requires AllocatableElement<Alloc, T, const T&> void push_front(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_front(T&& x);

```

```

requires AllocatableElement<Alloc, T, const T&> void push_back(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_back(T&& x);

template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...> && MoveAssignable<T>
    iterator emplace(const_iterator position, Args&&... args);

requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
    iterator insert(const_iterator position, const T& x);
requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
    iterator insert(const_iterator position, T&& x);
requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
    void insert(const_iterator position, size_type n, const T& x);
template <class InputIterator InputIterator Iter>
    requires AllocatableElement<Alloc, T, Iter::reference> && MoveAssignable<T>
    void insert(const_iterator position, InputIteratorIter first, InputIteratorIter last);
requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
    void insert(const_iterator position, initializer_list<T>);

void pop_front();
void pop_back();

requires MoveAssignable<T> iterator erase(const_iterator position);
requires MoveAssignable<T> iterator erase(const_iterator first, const_iterator last);
void swap(deque<T, Allocator>&&);
void clear();
};

template <class EqualityComparable T, class Allocator>
    bool operator==(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
template <class LessThanComparable T, class Allocator>
    bool operator< (const deque<T, Allocator>& x, const deque<T, Allocator>& y);
template <class EqualityComparable T, class Allocator>
    bool operator!=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
template <class LessThanComparable T, class Allocator>
    bool operator> (const deque<T, Allocator>& x, const deque<T, Allocator>& y);
template <class LessThanComparable T, class Allocator>
    bool operator>= (const deque<T, Allocator>& x, const deque<T, Allocator>& y);
template <class LessThanComparable T, class Allocator>
    bool operator<= (const deque<T, Allocator>& x, const deque<T, Allocator>& y);

// specialized algorithms:
template <class ValueType T, class Allocator>
    void swap(deque<T, Allocator>& x, deque<T, Allocator>& y);
template <class ValueType T, class Allocator>
    void swap(deque<T, Allocator>&& x, deque<T, Allocator>& y);
template <class ValueType T, class Allocator>
    void swap(deque<T, Allocator>& x, deque<T, Allocator>&& y);

template <class T, class Alloc

```

```

struct constructible_with_allocator_suffix<deque<T, Alloc>>
    : true_type { };
}

```

23.2.2.1 deque constructors, copy, and assignment

[deque.cons]

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

1 *Effects:* Constructs an empty deque, using the specified allocator.

2 *Complexity:* Constant.

```
requires AllocatableElement<Alloc, T> explicit deque(size_type n);
```

3 *Effects:* Constructs a deque with n default constructed elements.

4 ~~*Requires:* T shall be DefaultConstructible.~~

5 *Complexity:* Linear in n .

```
requires AllocatableElement<Alloc, T, const T&>
```

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

6 *Effects:* Constructs a deque with n copies of *value*, using the specified allocator.

7 ~~*Requires:* T shall be CopyConstructible.~~

8 *Complexity:* Linear in n .

```
template <class InputIterator, InputIterator Iter>
requires AllocatableElement<Alloc, T, Iter::reference>
deque(InputIterator first, InputIterator last,
      const Allocator& = Allocator());
```

9 *Effects:* Constructs a deque equal to the the range $[first, last)$, using the specified allocator.

10 *Complexity:* $distance(first, last)$.

```
template <class InputIterator, InputIterator Iter>
requires AllocatableElement<Alloc, T, Iter::reference>
&& HasAssign<T, Iter::reference>
void assign(InputIterator first, InputIterator last);
```

11 *Effects:*

```
erase(begin(), end());
insert(begin(), first, last);
```

```
requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
void assign(size_type n, const T& t);
```

12 *Effects:*

```
erase(begin(), end());
insert(begin(), n, t);
```

23.2.2.2 deque capacity

[deque.capacity]

requires AllocatableElement<Alloc, T>

void resize(size_type sz);

1 *Effects:* If `sz < size()`, equivalent to `erase(begin() + sz, end())`; If `size() < sz`, appends `sz - size()` default constructed elements to the sequence.

2 ~~*Requires: T shall be DefaultConstructible.*~~

requires AllocatableElement<Alloc, T, const T&>

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

3 *Effects:*

```
    if (sz > size())
        insert(end(), sz-size(), c);
    else if (sz < size())
        erase(begin()+sz, end());
    else
        ; // do nothing
```

4 ~~*Requires: T shall be CopyConstructible.*~~

23.2.2.3 deque modifiers

[deque.modifiers]

requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>

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

requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>

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

requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>

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

template <class InputIterator InputIterator Iter>

requires AllocatableElement<Alloc, T, Iter::reference> && MoveAssignable<T>void insert(const_iterator position,
InputIterator first, InputIterator last);

template <class... Args>

requires AllocatableElement<Alloc, T, Args&&...>

void emplace_front(Args&&... args);

template <class... Args>

requires AllocatableElement<Alloc, T, Args&&...>

void emplace_back(Args&&... args);

template <class... Args>

requires AllocatableElement<Alloc, T, Args&&...> && MoveAssignable<T>

iterator emplace(const_iterator position, Args&&... args);

requires AllocatableElement<Alloc, T, const T&> void push_front(const T& x);requires AllocatableElement<Alloc, T, T&&> void push_front(T&& x);requires AllocatableElement<Alloc, T, const T&> void push_back(const T& x);requires AllocatableElement<Alloc, T, T&&> void push_back(T&& x);

- 1 *Effects:* An insertion in the middle of the deque invalidates all the iterators and references to elements of the deque. An insertion at either end of the deque invalidates all the iterators to the deque, but has no effect on the validity of references to elements of the deque.
- 2 *Remarks:* If an exception is thrown other than by the copy constructor or assignment operator of T there are no effects.
- 3 *Complexity:* The complexity is linear in the number of elements inserted plus the lesser of the distances to the beginning and end of the deque. Inserting a single element either at the beginning or end of a deque always takes constant time and causes a single call to a constructor of T.

```
requires MoveAssignable<T> iterator erase(const_iterator position);
requires MoveAssignable<T> iterator erase(const_iterator first, const_iterator last);
```

- 4 *Effects:* An erase in the middle of the deque invalidates all the iterators and references to elements of the deque and the past-the-end iterator. An erase at the beginning of the deque invalidates only the iterators and the references to the erased elements. An erase at the end of the deque invalidates only the iterators and the references to the erased elements and the past-the-end iterator.
- 5 *Complexity:* The number of calls to the destructor is the same as the number of elements erased, but the number of the calls to the assignment operator is at most equal to the minimum of the number of elements before the erased elements and the number of elements after the erased elements.
- 6 *Throws:* Nothing unless an exception is thrown by the copy constructor or assignment operator of T.

23.2.2.4 deque specialized algorithms

[deque.special]

```
template <class ValueType T, class Allocator>
    void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
template <class ValueType T, class Allocator>
    void swap(deque<T,Allocator>&& x, deque<T,Allocator>& y);
template <class ValueType T, class Allocator>
    void swap(deque<T,Allocator>& x, deque<T,Allocator>&& y);
```

- 1 *Effects:*
`x.swap(y);`

23.2.3 Class template forward_list

[forwardlist]

- 1 A forward_list is a container that supports forward iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Fast random access to list elements is not supported. [*Note:* It is intended that forward_list have zero space or time overhead relative to a hand-written C-style singly linked list. Features that would conflict with that goal have been omitted. — end note]
- 2 A forward_list satisfies all of the requirements of a container (table 90), except that the size() member function is not provided. Descriptions are provided here only for operations on forward_list that are not described in that table or for operations where there is additional semantic information.

```

namespace std {
    template <class ValueType T, class Allocator Allocator = allocator<T> >
    requires NothrowDestructible<T>
    class forward_list {
    public:
        // types:
        typedef typename Allocator::reference reference;
        typedef typename Allocator::const_reference const_reference;
        typedef implementation-defined iterator;           // See 23.1
        typedef implementation-defined const_iterator;     // See 23.1
        typedef implementation-defined size_type;          // See 23.1
        typedef implementation-defined difference_type;     // See 23.1
        typedef T value_type;
        typedef Allocator allocator_type;
        typedef typename Allocator::pointer pointer;
        typedef typename Allocator::const_pointer const_pointer;

        // 23.2.3.1 construct/copy/destroy:
        explicit forward_list(const Allocator& = Allocator());
        requires AllocatableElement<Alloc, T>
        explicit forward_list(size_type n);
        requires AllocatableElement<Alloc, T, const T&>
        forward_list(size_type n, const T& value,
                     const Allocator& = Allocator());
        template <class InputIterator InputIterator Iter>
        AllocatableElement<Alloc, T, Iter::reference>
        forward_list(InputIterator first, InputIterator last,
                     const Allocator& = Allocator());
        requires AllocatableElement<Alloc, T, const T&>
        forward_list(const forward_list<T, Allocator>& x);
        requires AllocatableElement<Alloc, T, T&> forward_list(forward_list<T, Allocator>&& x);
        requires AllocatableElement<Alloc, T, const T&>
        forward_list(initializer_list<T>, const Allocator& = Allocator());
        ~forward_list();
        requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
        forward_list<T, Allocator>& operator=(const forward_list<T, Allocator>& x);
        requires AllocatableElement<Alloc, T, T&> && MoveAssignable<T>
        forward_list<T, Allocator>& operator=(forward_list<T, Allocator>&& x);
        requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
        forward_list<T, Allocator> operator=(initializer_list<T>);
        template <class InputIterator InputIterator Iter>
        requires AllocatableElement<Alloc, T, Iter::reference>
        && HasAssign<T, Iter::reference>
        void assign(InputIterator first, InputIterator last);
        requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
        void assign(size_type n, const T& t);
        requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
        void assign(initializer_list<T>);
        allocator_type get_allocator() const;
    };
}

```

```

// 23.2.3.2 iterators:
iterator before_begin();
const_iterator before_begin() const;
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

const_iterator cbegin() const;
const_iterator cbefore_begin() const;
const_iterator cend() const;

// capacity:
bool empty() const;
size_type max_size() const;

// 23.2.3.3 element access:
reference front();
const_reference front() const;

// 23.2.3.4 modifiers:
template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    void emplace_front(Args&&... args);
requires AllocatableElement<Alloc, T, const T&> void push_front(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_front(T&& x);
void pop_front();

template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    iterator emplace_after(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, T, const T&>
    iterator insert_after(const_iterator position, const T& x);
requires AllocatableElement<Alloc, T, T&&>
    iterator insert_after(const_iterator position, T&& x);
requires AllocatableElement<Alloc, T, const T&>
    void insert_after(const_iterator position, initializer_list<T> il);

requires AllocatableElement<Alloc, T, const T&>
    void insert_after(const_iterator position, size_type n, const T& x);
template <class InputIterator InputIterator Iter>
    requires AllocatableElement<Alloc, T, Iter::reference>
    void insert_after(const_iterator position, InputIteratorIter first, InputIteratorIter last);

iterator erase_after(const_iterator position);
iterator erase_after(const_iterator position, iterator last);
void swap(forward_list<T, Allocator>&&);

requires AllocatableElement<Alloc, T> void resize(size_type sz);
requires AllocatableElement<Alloc, T, const T&> void resize(size_type sz, value_type c);

```

```

void clear();

// 23.2.3.5 forward_list operations:
void splice_after(const_iterator position, forward_list<T,Allocator>&& x);
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 first, const_iterator last);

requires EqualityComparable<T> void remove(const T& value);
template <class Predicate<auto, T> Predicate> void remove_if(Predicate pred);

requires EqualityComparable<T> void unique();
template <class EquivalenceRelation<auto, T> BinaryPredicate>
    void unique(BinaryPredicate binary_pred);

requires LessThanComparable<T> void merge(forward_list<T,Allocator>&& x);
template <class StrictWeakOrder<auto, T> Compare>
    void merge(forward_list<T,Allocator>&& x, Compare comp);

requires LessThanComparable<T> void sort();
template <class StrictWeakOrder<auto, T> Compare> void sort(Compare comp);

void reverse();
};

// Comparison operators
template <class EqualityComparable T, class Allocator>
    bool operator==(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
template <class LessThanComparable T, class Allocator>
    bool operator< (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
template <class EqualityComparable T, class Allocator>
    bool operator!=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
template <class LessThanComparable T, class Allocator>
    bool operator> (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
template <class LessThanComparable T, class Allocator>
    bool operator>=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
template <class LessThanComparable T, class Allocator>
    bool operator<=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);

// 23.2.3.6 specialized algorithms:
template <class ValueType T, class Allocator>
    void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
template <class ValueType T, class Allocator>
    void swap(forward_list<T,Allocator>&& x, forward_list<T,Allocator>& y);
template <class ValueType T, class Allocator>
    void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>&& y);
}

```

23.2.3.1 forward_list constructors, copy, assignment**[forwardlist.cons]**

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

1 *Effects*: Constructs an empty forward_list object using the specified allocator.

2 *Complexity*: Constant.

```
requires AllocatableElement<Alloc, T>  
explicit forward_list(size_type n);
```

3 *Effects*: Constructs a forward_list object with n default constructed elements.

4 ~~*Requires*: T shall be DefaultConstructible.~~

5 *Complexity*: Linear in n.

```
requires AllocatableElement<Alloc, T, const T&>  
forward_list(size_type n, const T& value, const Allocator& = Allocator());
```

6 *Effects*: Constructs a forward_list object with n copies of value using the specified allocator.

7 ~~*Requires*: T shall be CopyConstructible.~~

8 *Complexity*: Linear in n.

```
template <class InputIterator InputIterator Iter>  
AllocatableElement<Alloc, T, Iter::reference>  
forward_list(InputIteratorIter first, InputIteratorIter last,  
            const Allocator& = Allocator());
```

9 *Effects*: Constructs a forward_list object equal to the range [first,last).

10 *Complexity*: Linear in distance(first, last).

```
template <class InputIterator InputIterator Iter>  
requires AllocatableElement<Alloc, T, Iter::reference>  
&& HasAssign<T, Iter::reference>  
void assign(InputIteratorIter first, InputIteratorIter last);
```

11 *Effects*: clear(); insert_after(before_begin(), first, last);

```
AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>  
void assign(size_type n, const T& t);
```

12 *Effects*: clear(); insert_after(before_begin(), n, t);

23.2.3.2 forward_list iterators**[forwardlist.iter]**

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

1 *Returns*: A non-dereferenceable iterator that, when incremented, is equal to the iterator returned by begin().

23.2.3.3 forward_list element access

[forwardlist.access]

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

1 *Returns:**begin()

23.2.3.4 forward_list modifiers

[forwardlist.modifiers]

1 None of the overloads of insert_after shall affect the validity of iterators and reference, and erase_after shall invalidate only the iterators and references to the erased elements. If an exception is thrown during insert_after there shall be no effect. Insertion of n elements into a forward_list is linear in n, and the number of calls to the copy or move constructor of T is exactly equal to n. Erasing n elements from a forward_list is linear time in n and the number of calls to the destructor of type T is exactly equal to n.

```
template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    void emplace_front(Args&&... args);
```

2 *Effects:* Inserts an object of type value_type constructed with value_type(forward<Args>(args)...) at the beginning of the list.

```
requires AllocatableElement<Alloc, T, const T&> void push_front(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_front(T&& x);
```

3 *Effects:* Inserts a copy of x at the beginning of the list.

```
void pop_front();
```

4 *Effects:* erase_after(before_begin())

```
requires AllocatableElement<Alloc, T, const T&>
    iterator insert_after(const_iterator position, const T& x);
requires AllocatableElement<Alloc, T, T&&>
    iterator insert_after(const_iterator position, T&& x);
```

5 *Requires:* position is dereferenceable or equal to before_begin().

6 *Effects:* Inserts a copy of x after position.

7 *Returns:* An iterator pointing to the copy of x.

```
requires AllocatableElement<Alloc, T, const T&>
    void insert_after(const_iterator position, size_type n, const T& x);
```

8 *Requires:* position is dereferenceable or equal to before_begin().

9 *Effects:* Inserts n copies of x after position.

```
template <class InputIterator InputIterator Iter>
    requires AllocatableElement<Alloc, T, Iter::reference>
    void insert_after(const_iterator position, InputIteratorIter first, InputIteratorIter last);
```

10 *Requires:* position is dereferenceable or equal to before_begin(). first and last are not iterators in *this.

11 *Effects:* Inserts copies of elements in [first,last) after position.

requires AllocatableElement<Alloc, T, const T&>

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

12 *Effects:* insert_after(p, s.begin(), s.end()).

template <class... Args>

requires AllocatableElement<Alloc, T, Args&&...>

iterator emplace_after(const_iterator position, Args&&... args);

13 *Requires:* position is dereferenceable or equal to before_begin().

14 *Effects:* Inserts an object of type value_type constructed with value_type(forward<Args>(args)...) after position.

iterator erase_after(const_iterator position);

15 *Requires:* The iterator following position is dereferenceable.

16 *Effects:* Erases the element pointed to by the iterator following position.

17 *Returns:* An iterator pointing to the element following the one that was erased, or end() if no such element exists.

iterator erase_after(const_iterator position, iterator last);

18 *Requires:* All iterators in the range [position,last) are dereferenceable.

19 *Effects:* Erases the elements in the range [position,last).

20 *Returns:* last

requires AllocatableElement<Alloc, T> void resize(size_type sz);

requires AllocatableElement<Alloc, T, const T&> void resize(size_type sz, value_type c);

21 *Effects:* If sz < distance(begin(), end()), erases the last distance(begin(), end()) - sz elements from the list. Otherwise, inserts sz - distance(begin(), end()) elements at the end of the list. For the first signature the inserted elements are default constructed, and for the second signature they are copies of c.

void clear();

22 *Effects:* Erases all elements in the range [begin(), end()).

23.2.3.5 forward_list operations

[forwardlist.ops]

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

1 *Requires:* position is dereferenceable or equal to before_begin(). &x != this.

2 *Effects:* Inserts the contents of x before position, and x becomes empty. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

3 *Throws:*Nothing.

4 *Complexity:* $\mathcal{O}(1)$

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

5 *Requires:*position is dereferenceable or equal to before_begin(). The iterator following i is a dereferenceable iterator in x.

6 *Effects:*Inserts the element following i into *this, following position, and removes it from x. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

7 *Throws:*Nothing.

8 *Complexity:* $\mathcal{O}(1)$

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

9 *Requires:*position is dereferenceable or equal to before_begin(). (first,last) is a valid range in x, and all iterators in the range (first,last) are dereferenceable. position is not an iterator in the range (first, last).

10 *Effects:*Inserts elements in the range (first,last) after position and removes the elements from x. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

```
requires EqualityComparable<T> void remove(const T& value);
```

```
template <class Predicate<auto, T> Predicate> void remove_if(Predicate pred);
```

11 *Effects:*Erases all the elements in the list referred by a list iterator i for which the following conditions hold: *i == value (for remove()), pred(*i) is true (for remove_if()). This operation shall be stable: the relative order of the elements that are not removed is the same as their relative order in the original list.

12 *Throws:*Nothing unless an exception is thrown by the equality comparison or the predicate.

13 *Complexity:*Exactly distance(begin(), end()) applications of the corresponding predicate.

```
requires EqualityComparable<T> void unique();
```

```
template <class EquivalenceRelation<auto, T> BinaryPredicate>
void unique(BinaryPredicate pred);
```

14 *Effects:*Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator i in the range [first + 1,last) for which *i == *(i-1) (for the version with no arguments) or pred(*i, *(i - 1)) (for the version with a predicate argument) holds.

15 *Throws:*Nothing unless an exception is thrown by the equality comparison or the predicate.

16 *Complexity:*If the range [first,last) is not empty, exactly (last - first) - 1 applications of the corresponding predicate, otherwise no applications of the predicate.

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

17 *Requires:* ~~comp defines a strict weak ordering (??), and~~ *this and x are both sorted according to ~~this ordering~~ the strict weak ordering defined by operator< or comp.

18 *Effects:* Merges x into *this. This operation shall be stable: for equivalent elements in the two lists, the elements from *this shall always precede the elements from x. x is empty after the merge. If an exception is thrown other than by a comparison there are no effects.

19 *Complexity:* At most $\text{size}() + x.\text{size}() - 1$ comparisons.

```
requires LessThanComparable<T> void sort();
template <class StrictWeakOrder<auto, T> Compare> void sort(Compare comp);
```

20 *Requires:* ~~operator< (for the version with no arguments) or comp (for the version with a comparison argument) defines a strict weak ordering (??).~~

21 *Effects:* Sorts the list according to the operator< or the comp function object. This operation shall be stable: the relative order of the equivalent elements is preserved. If an exception is thrown the order of the elements in *this is unspecified.

22 *Complexity:* Approximately $N \log N$ comparisons, where N is $\text{distance}(\text{begin}(), \text{end}())$.

```
void reverse();
```

23 *Effects:* Reverses the order of the elements in the list.

24 *Throws:* Nothing.

25 *Complexity:* Linear time.

23.2.3.6 forward_list specialized algorithms

[forwardlist.spec]

```
template <class ValueType T, class Allocator>
void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
template <class ValueType T, class Allocator>
void swap(forward_list<T,Allocator>&& x, forward_list<T,Allocator>& y);
template <class ValueType T, class Allocator>
void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>&& y);
```

1 *Effects:* $x.\text{swap}(y)$

23.2.4 Class template list

[list]

1 A list is a sequence container that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Unlike vectors (23.2.6) and deques (23.2.2), fast random access to list elements is not supported, but many algorithms only need sequential access anyway.

- 2 A list satisfies all of the requirements of a container, of a reversible container (given in two tables in 23.1), of a sequence container, including most of the the optional sequence container requirements (23.1.1), and of an allocator-aware container (Table ??). The exceptions are the operator[] and at member functions, which are not provided.¹⁾ Descriptions are provided here only for operations on list that are not described in one of these tables or for operations where there is additional semantic information.

```

namespace std {
    template <class ValueType T, class Allocator Allocator = allocator<T> >
        requires NothrowDestructible<T>
        class list {
        public:
            // types:
            typedef typename Allocator::reference          reference;
            typedef typename Allocator::const_reference    const_reference;
            typedef implementation-defined                 iterator;           // See 23.1
            typedef implementation-defined                 const_iterator;     // See 23.1
            typedef implementation-defined                 size_type;           // See 23.1
            typedef implementation-defined                 difference_type;      // See 23.1
            typedef T                                       value_type;
            typedef Allocator                             allocator_type;
            typedef typename Allocator::pointer            pointer;
            typedef typename Allocator::const_pointer      const_pointer;
            typedef reverse_iterator<iterator>             reverse_iterator;
            typedef reverse_iterator<const_iterator>       const_reverse_iterator;

            // 23.2.4.1 construct/copy/destroy:
            explicit list(const Allocator& = Allocator());
            requires AllocatableElement<Alloc, T> explicit list(size_type n);
            requires AllocatableElement<Alloc, T, const T&>
                list(size_type n, const T& value, const Allocator& = Allocator());
            template <class InputIterator InputIterator Iter>
                requires AllocatableElement<Alloc, T, Iter::reference>
                list(InputIterator first, InputIterator last, const Allocator& = Allocator());
            requires AllocatableElement<Alloc, T, const T&> list(const list<T, Allocator>& x);
            requires AllocatableElement<Alloc, T, T&&> list(list&& x);
            requires AllocatableElement<Alloc, T, const T&> list(const list&, const Allocator&);
            requires AllocatableElement<Alloc, T, T&&> list(list&&, const Allocator&);
            requires AllocatableElement<Alloc, T, const T&>
                list(initializer_list<T>, const Allocator& = Allocator());
            ~list();
            requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
                list<T, Allocator>& operator=(const list<T, Allocator>& x);
            requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
                list<T, Allocator>& operator=(list<T, Allocator>&& x);
            requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
                list<T, Allocator>& operator=(initializer_list<T>);
            template <class InputIterator InputIterator Iter>
                requires AllocatableElement<Alloc, T, Iter::reference>

```

¹⁾ These member functions are only provided by containers whose iterators are random access iterators.

```

    && HasAssign<T, Iter::reference>
    void assign(InputIteratorIter first, InputIteratorIter last);
    requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
    void assign(size_type n, const T& t);
    requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
    void assign(initializer_list<T>);
    allocator_type get_allocator() const;

// iterators:
iterator          begin();
const_iterator    begin() const;
iterator          end();
const_iterator    end() const;
reverse_iterator  rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator  rend();
const_reverse_iterator rend() const;

const_iterator    cbegin() const;
const_iterator    cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// 23.2.4.2 capacity:
bool              empty() const;
size_type         size() const;
size_type         max_size() const;
requires AllocatableElement<Alloc, T> void resize(size_type sz);
requires AllocatableElement<Alloc, T, const T&> void resize(size_type sz, const T& c);

// element access:
reference         front();
const_reference   front() const;
reference         back();
const_reference   back() const;

// 23.2.4.3 modifiers:
template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    void emplace_front(Args&&... args);
void pop_front();
template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    void emplace_back(Args&&... args);
void pop_back();

requires AllocatableElement<Alloc, T, const T&> void push_front(const T& x);
requires AllocatableElement<Alloc, T, T&&> void push_front(T&& x);
requires AllocatableElement<Alloc, T, const T&> void push_back(const T& x);
requires AllocatableElement<Alloc, T, T&&> void push_back(T&& x);

```

```

template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    iterator emplace(const_iterator position, Args&&... args);
    requires AllocatableElement<Alloc, T, const T&>
    iterator insert(const_iterator position, const T& x);
    requires AllocatableElement<Alloc, T, T&&>
    iterator insert(const_iterator position, T&& x);
    requires AllocatableElement<Alloc, T, const T&>
    void insert(const_iterator position, size_type n, const T& x);
template <class InputIterator, InputIterator Iter>
    requires AllocatableElement<Alloc, T, Iter::reference>
    void insert(const_iterator position, InputIterator first,
               InputIterator last);
    requires AllocatableElement<Alloc, T, const T&>
    void insert(const_iterator position, initializer_list<T> il);

iterator erase(const_iterator position);
iterator erase(const_iterator position, const_iterator last);
void swap(list<T, Allocator>&&);
void clear();

// 23.2.4.4 list operations:
void splice(const_iterator position, list<T, Allocator>&& x);
void splice(const_iterator position, list<T, Allocator>&& x, const_iterator i);
void splice(const_iterator position, list<T, Allocator>&& x,
           const_iterator first, const_iterator last);

    requires EqualityComparable<T> void remove(const T& value);
template <class Predicate<auto, T> Predicate> void remove_if(Predicate pred);

    requires EqualityComparable<T> void unique();
template <class EquivalenceRelation<auto, T> BinaryPredicate>
    void unique(BinaryPredicate binary_pred);

    requires LessThanComparable<T> void merge(list<T, Allocator>&& x);
template <class StrictWeakOrder<auto, T> Compare>
    void merge(list<T, Allocator>&& x, Compare comp);

    requires LessThanComparable<T> void sort();
template <class StrictWeakOrder<auto, T> Compare>
    void sort(Compare comp);

void reverse();
};

template <class EqualityComparable T, class Allocator>
    bool operator==(const list<T, Allocator>& x, const list<T, Allocator>& y);
template <class LessThanComparable T, class Allocator>
    bool operator< (const list<T, Allocator>& x, const list<T, Allocator>& y);

```

```

template <class EqualityComparable T, class Allocator>
    bool operator!=(const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class LessThanComparable T, class Allocator>
    bool operator> (const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class LessThanComparable T, class Allocator>
    bool operator>=(const list<T,Allocator>& x, const list<T,Allocator>& y);
template <class LessThanComparable T, class Allocator>
    bool operator<=(const list<T,Allocator>& x, const list<T,Allocator>& y);

// specialized algorithms:
template <class ValueType T, class Allocator>
    void swap(list<T,Allocator>& x, list<T,Allocator>& y);
template <class ValueType T, class Allocator>
    void swap(list<T,Allocator>&& x, list<T,Allocator>& y);
template <class ValueType T, class Allocator>
    void swap(list<T,Allocator>& x, list<T,Allocator>&& y);

template <class T, class Alloc>
    struct constructible_with_allocator_suffix<list<T, Alloc>>
        : true_type { };
}

```

23.2.4.1 list constructors, copy, and assignment

[list.cons]

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

1 *Effects:* Constructs an empty list, using the specified allocator.

2 *Complexity:* Constant.

```
requires AllocatableElement<Alloc, T> explicit list(size_type n);
```

3 *Effects:* Constructs a list with n default constructed elements.

4 ~~*Requires:* T shall be DefaultConstructible.~~

5 *Complexity:* Linear in n.

```
requires AllocatableElement<Alloc, T, const T&>
list(size_type n, const T& value,
    const Allocator& = Allocator());
```

6 *Effects:* Constructs a list with n copies of value, using the specified allocator.

7 ~~*Requires:* T shall be CopyConstructible.~~

8 *Complexity:* Linear in n.

```
template <class InputIterator InputIterator Iter>
requires AllocatableElement<Alloc, T, Iter::reference>
list(InputIterator first, InputIterator last, const Allocator& = Allocator());
```

9 *Effects:* Constructs a list equal to the range $[first, last)$.

10 *Complexity:* Linear in `distance(first, last)`.

```
template <class InputIterator, InputIterator Iter>
    requires AllocatableElement<Alloc, T, Iter::reference>
           && HasAssign<T, Iter::reference>
void assign(InputIterator Iter first, InputIterator Iter last);
```

11 *Effects:* Replaces the contents of the list with the range `[first, last)`.

```
erase(begin(), end());
insert(begin(), n, t);
```

```
requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
void assign(size_type n, const T& t);
```

12 *Effects:* Replaces the contents of the list with n copies of t .

23.2.4.2 list capacity

[list.capacity]

```
requires AllocatableElement<Alloc, T> void resize(size_type sz);
```

1 *Effects:* If `sz < size()`, equivalent to `list<T>::iterator it = begin(); advance(it, sz); erase(it, end());`. If `size() < sz`, appends `sz - size()` default constructed elements to the sequence.

2 ~~*Requires:* T shall be DefaultConstructible.~~

```
requires AllocatableElement<Alloc, T, const T&> void resize(size_type sz, const T& c);
```

3 *Effects:*

```
if (sz > size())
    insert(end(), sz-size(), c);
else if (sz < size()) {
    iterator i = begin();
    advance(i, sz);
    erase(i, end());
}
else
    ; // do nothing
```

4 ~~*Requires:* T shall be CopyConstructible.~~

23.2.4.3 list modifiers

[list.modifiers]

```
requires AllocatableElement<Alloc, T, const T&>
    iterator insert(const_iterator position, const T& x);
requires AllocatableElement<Alloc, T, T&&>
    iterator insert(const_iterator position, T&& x);
requires AllocatableElement<Alloc, T, const T&>
    void insert(const_iterator position, size_type n, const T& x);
```

```

template <class InputIterator InputIterator Iter>
    requires AllocatableElement<Alloc, T, Iter::reference>
    void insert(const_iterator position, InputIteratorIter first,
               InputIteratorIter last);

template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    void emplace_front(Args&&... args);
template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    void emplace_back(Args&&... args);
template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    iterator emplace(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, T, const T&> void push_front(const T& x);
requires AllocatableElement<Alloc, T, T&&> void push_front(T&& x);
requires AllocatableElement<Alloc, T, const T&> void push_back(const T& x);
requires AllocatableElement<Alloc, T, T&&> void push_back(T&& x);

```

- 1 *Remarks:* Does not affect the validity of iterators and references. If an exception is thrown there are no effects.
- 2 *Complexity:* Insertion of a single element into a list takes constant time and exactly one call to a constructor of T. Insertion of multiple elements into a list is linear in the number of elements inserted, and the number of calls to the copy constructor or move constructor of T is exactly equal to the number of elements inserted.

```

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

void pop_front();
void pop_back();
void clear();

```

- 3 *Effects:* Invalidates only the iterators and references to the erased elements.
- 4 *Throws:* Nothing.
- 5 *Complexity:* Erasing a single element is a constant time operation with a single call to the destructor of T. Erasing a range in a list is linear time in the size of the range and the number of calls to the destructor of type T is exactly equal to the size of the range.

23.2.4.4 list operations

[list.ops]

- 1 Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for them.²⁾
- 2 list provides three splice operations that destructively move elements from one list to another. The behavior of splice operations is undefined if `get_allocator() != x.get_allocator()`.

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

²⁾As specified in ??, the requirements in this clause apply only to lists whose allocators compare equal.

3 *Requires:* $\&x \neq \text{this}$.

4 *Effects:* Inserts the contents of x before position and x becomes empty. Pointers and references to the moved elements of x now refer to those same elements but as members of *this . Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this , not into x .

5 *Throws:* Nothing

6 *Complexity:* Constant time.

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

7 *Effects:* Inserts an element pointed to by i from list x before position and removes the element from x . The result is unchanged if $\text{position} == i$ or $\text{position} == ++i$. Pointers and references to *i continue to refer to this same element but as a member of *this . Iterators to *i (including i itself) continue to refer to the same element, but now behave as iterators into *this , not into x .

8 *Throws:* Nothing

9 *Requires:* i is a valid dereferenceable iterator of x .

10 *Complexity:* Constant time.

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

11 *Effects:* Inserts elements in the range $[\text{first}, \text{last})$ before position and removes the elements from x .

12 *Requires:* $[\text{first}, \text{last})$ is a valid range in x . The result is undefined if position is an iterator in the range $[\text{first}, \text{last})$. Pointers and references to the moved elements of x now refer to those same elements but as members of *this . Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this , not into x .

13 *Throws:* Nothing

14 *Complexity:* Constant time if $\&x == \text{this}$; otherwise, linear time.

```
requires EqualityComparable<T> void remove(const T& value);
```

```
template <class Predicate<auto, T> Predicate> void remove_if(Predicate pred);
```

15 *Effects:* Erases all the elements in the list referred by a list iterator i for which the following conditions hold: $\text{*i} == \text{value}$, $\text{pred}(\text{*i}) \neq \text{false}$.

16 *Throws:* Nothing unless an exception is thrown by $\text{*i} == \text{value}$ or $\text{pred}(\text{*i}) \neq \text{false}$.

17 *Remarks:* Stable.

18 *Complexity:* Exactly $\text{size}()$ applications of the corresponding predicate.

```
requires EqualityComparable<T> void unique();
```

```
template <class EquivalenceRelation<auto, T> BinaryPredicate> void unique(BinaryPredicate binary_pred);
```

19 *Effects:* Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator i in the range $[\text{first} + 1, \text{last})$ for which $\text{*i} == \text{*(i-1)}$ (for the version of unique with no arguments) or $\text{pred}(\text{*i}, \text{*(i - 1)})$ (for the version of unique with a predicate argument) holds.

- 20 *Throws:* Nothing unless an exception is thrown by `*i == *(i-1)` or `pred(*i, *(i - 1))`
- 21 *Complexity:* If the range `[first, last)` is not empty, exactly `(last - first) - 1` applications of the corresponding predicate, otherwise no applications of the predicate.

```
requires LessThanComparable<T> void merge(list<T,Allocator>&& x);
template <class StrictWeakOrder<auto, T> Compare>
void merge(list<T,Allocator>&& x, Compare comp);
```

- 22 *Requires:* ~~`comp` shall define a strict weak ordering (??), and~~ both the list and the argument list shall be sorted according to ~~`this ordering operator<` or `comp`.~~
- 23 *Effects:* If `(&x == this)` does nothing; otherwise, merges the two sorted ranges `[begin(), end())` and `[x.begin(), x.end())`. The result is a range in which the elements will be sorted in non-decreasing order according to the ordering defined by `comp`; that is, for every iterator `i`, in the range other than the first, the condition `comp(*i, *(i - 1))` will be false.
- 24 *Remarks:* Stable. If `(&x != this)` the range `[x.begin(), x.end())` is empty after the merge.
- 25 *Complexity:* At most `size() + x.size() - 1` applications of `comp` if `(&x != this)`; otherwise, no applications of `comp` are performed. If an exception is thrown other than by a comparison there are no effects.

```
void reverse();
```

- 26 *Effects:* Reverses the order of the elements in the list.
- 27 *Throws:* Nothing.
- 28 *Complexity:* Linear time.

```
requires LessThanComparable<T> void sort();
template <class StrictWeakOrder<auto, T> Compare> void sort(Compare comp);
```

- 29 *Requires:* ~~`operator<` (for the first version) or `comp` (for the second version) shall define a strict weak ordering (??).~~
- 30 *Effects:* Sorts the list according to the `operator<` or a `Compare` function object.
- 31 *Remarks:* Stable.
- 32 *Complexity:* Approximately $N \log(N)$ comparisons, where $N == \text{size}()$.

23.2.4.5 list specialized algorithms

[list.special]

```
template <class ValueType T, class Allocator>
void swap(list<T,Allocator>& x, list<T,Allocator>& y);
template <class ValueType T, class Allocator>
void swap(list<T,Allocator>&& x, list<T,Allocator>& y);
template <class ValueType T, class Allocator>
void swap(list<T,Allocator>& x, list<T,Allocator>&& y);
```

- 1 *Effects:*
- ```
 x.swap(y);
```

## 23.2.5 Container adaptors

[container.adaptors]

- 1 The container adaptors each take a Container template parameter, and each constructor takes a Container reference argument. This container is copied into the Container member of each adaptor. If the container takes an allocator, then a compatible allocator may be passed in to the adaptor's constructor. Otherwise, normal copy or move construction is used for the container argument. [Note: it is not necessary for an implementation to distinguish between the one-argument constructor that takes a Container and the one-argument constructor that takes an allocator\_type. Both forms use their argument to construct an instance of the container. — end note ]

## 23.2.5.1 Class template queue

[queue]

- 1 Any sequence container ~~supporting operations front(), back(), push\_back() and pop\_front()~~ meeting the requirements of the FrontInsertionContainer and BackInsertionContainer concepts can be used to instantiate queue. In particular, list (23.2.4) and deque (23.2.2) can be used.

## 23.2.5.1.1 queue definition

[queue.defn]

```

namespace std {
 template <class ObjectType T, class Container = deque<T> >
 requires QueueLikeContainer<Cont>
 && SameType<T, Cont::value_type>
 && NothrowDestructible<Cont>
 class queue {
 public:
 typedef typename Container::value_type value_type;
 typedef typename Container::reference reference;
 typedef typename Container::const_reference const_reference;
 typedef typename Container::size_type size_type;
 typedef Container container_type;
 protected:
 Container c;

 public:
 requires CopyConstructible<Cont> explicit queue(const Container&);
 requires MoveConstructible<Cont> explicit queue(Container&& = Container());
 requires MoveConstructible<Cont> queue(queue&& q) : c(move(q.c)) {}
 template <class Alloc>
 requires Constructible<Cont, const Alloc>
 explicit queue(const Alloc&);
 template <class Alloc>
 requires Constructible<Cont, const Cont&, const Alloc>
 queue(const Container&, const Alloc&);
 template <class Alloc>
 requires Constructible<Cont, Cont&&, const Alloc>
 queue(Container&&, const Alloc&);
 template <class Alloc>
 requires Constructible<Cont, Cont&&, const Alloc>
 queue(queue&&, const Alloc&);
 requires MoveAssignable<Cont> queue& operator=(queue&& q)
 { c = move(q.c); return *this; }
 };

```

```

bool empty() const { return e.empty(c); }
size_type size() const { return e.size(c); }
reference front() { return e.front(c); }
const_reference front() const { return e.front(c); }
reference back() { return e.back(c); }
const_reference back() const { return e.back(c); }
void push(const value_type& x) { e.push_back(c, x); }
void push(value_type&& x) { e.push_back(c, move(x)); }
template <class... Args>
 requires BackEmplacementContainer<Cont, Args&&...>
void emplace(Args&&... args)
 { e.emplace_back(c, forward<Args>(args)...); }
void pop() { e.pop_front(c); }
 requires Swappable<Cont>
void swap(queue&& q) { e.swap(c, q.c); }
};

template <class T, class EqualityComparable Container>
 bool operator==(const queue<T, Container>& x, const queue<T, Container>& y);
template <class T, class LessThanComparable Container>
 bool operator< (const queue<T, Container>& x, const queue<T, Container>& y);
template <class T, class EqualityComparable Container>
 bool operator!=(const queue<T, Container>& x, const queue<T, Container>& y);
template <class T, class LessThanComparable Container>
 bool operator> (const queue<T, Container>& x, const queue<T, Container>& y);
template <class T, class LessThanComparable Container>
 bool operator>=(const queue<T, Container>& x, const queue<T, Container>& y);
template <class T, class LessThanComparable Container>
 bool operator<=(const queue<T, Container>& x, const queue<T, Container>& y);

template <class ObjectType T, class Swappable Container>
 void swap(queue<T, Container>& x, queue<T, Container>& y);
template <class ObjectType T, class Swappable Container>
 void swap(queue<T, Container>&& x, queue<T, Container>& y);
template <class ObjectType T, class Swappable Container>
 void swap(queue<T, Container>& x, queue<T, Container>&& y);

template <class T, class Cont, class Alloc>
 requires UsesAllocator<Cont, Alloc>
 concept_map UsesAllocator<queue<T, Cont>, Alloc> { }

template <class T, class Container, class Alloc>
 struct uses_allocator<queue<T, Container>, Alloc>
 : uses_allocator<Container, Alloc>::type { };

template <class T, class Container>
 struct constructible_with_allocator_suffix<queue<T, Container>>
 : true_type { };
}

```

## 23.2.5.1.2 queue operators

[queue.ops]

```
template <class T, class EqualityComparable Container>
 bool operator==(const queue<T, Container>& x,
 const queue<T, Container>& y);
```

1     *Returns:* x.c == y.c.

```
template <class T, class EqualityComparable Container>
 bool operator!=(const queue<T, Container>& x,
 const queue<T, Container>& y);
```

2     *Returns:* x.c != y.c.

```
template <class T, class LessThanComparable Container>
 bool operator< (const queue<T, Container>& x,
 const queue<T, Container>& y);
```

3     *Returns:* x.c < y.c.

```
template <class T, class LessThanComparable Container>
 bool operator<=(const queue<T, Container>& x,
 const queue<T, Container>& y);
```

4     *Returns:* x.c <= y.c.

```
template <class T, class LessThanComparable Container>
 bool operator> (const queue<T, Container>& x,
 const queue<T, Container>& y);
```

5     *Returns:* x.c > y.c.

```
template <class T, class LessThanComparable Container>
 bool operator>=(const queue<T, Container>& x,
 const queue<T, Container>& y);
```

6     *Returns:* x.c >= y.c.

## 23.2.5.1.3 queue specialized algorithms

[queue.special]

```
template <class ObjectType T, class Swappable Container>
 void swap(queue<T, Container>& x, queue<T, Container>& y);
template <class ObjectType T, class Swappable Container>
 void swap(queue<T, Container>&& x, queue<T, Container>& y);
template <class ObjectType T, class Swappable Container>
 void swap(queue<T, Container>& x, queue<T, Container>&& y);
```

1     *Effects:* x.swap(y).

## 23.2.5.2 Class template priority\_queue

[priority.queue]

- 1 Any sequence container with random access iterator and ~~supporting operations `front()`, `push_back()` and `pop_back()`~~ that meets the requirements of the `BackInsertionContainer` concept can be used to instantiate `priority_queue`. In particular, `vector` (23.2.6) and `deque` (23.2.2) can be used. Instantiating `priority_queue` also involves supplying a function or function object for making priority comparisons; the library assumes that the function or function object defines a strict weak ordering (??).

```
namespace std {
 template <class ObjectType T, class StackLikeContainer Container = vector<T>,
 class StrictWeakOrder<auto, T> Compare = less<typename Container::value_type> >
 requires SameType<Container::value_type, T> && RandomAccessIterator<Container::iterator>
 && ShuffleIterator<Container::iterator> && CopyConstructible<Compare>
 && NothrowDestructible<Container>
 class priority_queue {
 public:
 typedef typename Container::value_type value_type;
 typedef typename Container::reference reference;
 typedef typename Container::const_reference const_reference;
 typedef typename Container::size_type size_type;
 typedef Container container_type;

 protected:
 Container c;
 Compare comp;

 public:
 requires CopyConstructible<Container> priority_queue(const Compare& x, const Container&);
 requires MoveConstructible<Container>
 explicit priority_queue(const Compare& x = Compare(), Container&& = Container());
 template <class InputIterator InputIterator Iter>
 requires CopyConstructible<Container> && RangeInsertionContainer<Container, Iter>
 priority_queue(InputIteratorIter first, InputIteratorIter last,
 const Compare& x, const Container&);
 template <class InputIterator InputIterator Iter>
 requires MoveConstructible<Container> && RangeInsertionContainer<Container, Iter>
 priority_queue(InputIteratorIter first, InputIteratorIter last,
 const Compare& x = Compare(), Container&& = Container());
 requires MoveConstructible<Container> priority_queue(priority_queue&&);
 requires MoveAssignable<Container> priority_queue& operator=(priority_queue&&);
 template <class Alloc>
 requires Constructible<Container, const Alloc&>
 explicit priority_queue(const Alloc&);
 template <class Alloc>
 requires Constructible<Container, const Alloc&>
 priority_queue(const Compare&, const Alloc&);
 template <class Alloc>
 requires Constructible<Container, Container, Alloc>
 priority_queue(const Compare&, const Container&, const Alloc&);
 template <class Alloc>
 requires Constructible<Container, Container&&, Alloc>
```

```

 priority_queue(const Compare&, Container&&, const Alloc&);
template <class Alloc>
 requires Constructible<Cont, Cont&&, Alloc>
 priority_queue(priority_queue&&, const Alloc&);

bool empty() const { return c.empty(c); }
size_type size() const { return c.size(c); }
const_reference top() const { return c.front()*begin(c); }
void push(const value_type& x);
void push(value_type&& x);
template <class... Args>
 requires BackEmplacementContainer<Cont, Args&&...>
 void emplace(Args&&... args);
void pop();
 requires Swappable<Cont>
 void swap(priority_queue&&);
};

 // no equality is provided
template <class ObjectType T, class Swappable Container, Swappable Compare>
 void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>& y);
template <class ObjectType T, class Swappable Container, Swappable Compare>
 void swap(priority_queue<T, Container, Compare>&& x, priority_queue<T, Container, Compare>& y);
template <class ObjectType T, class Swappable Container, Swappable Compare>
 void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>&& y);

template <class T, class Cont, class Compare, class Alloc>
 requires UsesAllocator<Cont, Alloc>
 concept_map UsesAllocator<priority_queue<T, Cont, Compare>, Alloc> { }

template <class T, class Container, class Compare, class Alloc>
 struct uses_allocator<priority_queue<T, Container, Compare>, Alloc>
 : uses_allocator<Container, Alloc>::type { };

template <class T, class Container, class Compare>
 struct constructible_with_allocator_suffix<
 priority_queue<T, Container, Compare>->
 : true_type { };
}

```

### 23.2.5.2.1 priority\_queue constructors

[priority\_queue.cons]

requires CopyConstructible<Cont> priority\_queue(const Compare& x, const Container& y);

requires MoveConstructible<Cont>

explicit priority\_queue(const Compare& x = Compare(), Container&& y = Container());

1 ~~Requires:  $\omega$ -shall define a strict weak ordering (??).~~

2 *Effects:* Initializes comp with x and c with y (copy constructing or move constructing as appropriate); calls make\_heap(~~c~~.begin(c), ~~c~~.end(c), comp).

```
template <class InputIterator InputIterator Iter>
 requires CopyConstructible<Cont> && RangeInsertionContainer<Cont, Iter>
 priority_queue(InputIteratorIter first, InputIteratorIter last,
 const Compare& x, const Container&);
template <class InputIterator InputIterator Iter>
 requires MoveConstructible<Cont> && RangeInsertionContainer<Cont, Iter>
 priority_queue(InputIteratorIter first, InputIteratorIter last,
 const Compare& x = Compare(), Container&& = Container());
```

3 *Requires:* ~~x~~ shall define a strict weak ordering (??).

4 *Effects:* Initializes comp with x and c with y (copy constructing or move constructing as appropriate); calls ~~e~~.insert(c, ~~e~~.end(c), first, last); and finally calls make\_heap(~~e~~.begin(c), ~~e~~.end(c), comp).

### 23.2.5.2.2 priority\_queue members

[priqueue.members]

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

1 *Effects:*

```
e.push_back(c, x);
push_heap(e.begin(c), e.end(c), comp);
```

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

2 *Effects:*

```
e.push_back(c, move(x));
push_heap(e.begin(c), e.end(c), comp);
```

```
template <class... Args>
 requires BackEmplacementContainer<Cont, Args&&...>
 void emplace(Args&&... args);
```

3 *Effects:*

```
e.emplace_back(c, forward<Args>(args)...);
push_heap(e.begin(c), e.end(c), comp);
```

```
void pop();
```

4 *Effects:*

```
pop_heap(e.begin(c), e.end(c), comp);
e.pop_back(c);
```

### 23.2.5.2.3 priority\_queue specialized algorithms

[priqueue.special]

```
template <class T, classSwappable Container, classSwappable Compare>
 void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>& y);
```

```

template <class T, classSwappable Container, classSwappable Compare>
 void swap(priority_queue<T, Container, Compare>&& x, priority_queue<T, Container, Compare>& y);
template <class T, classSwappable Container, classSwappable Compare>
 void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>&& y);
1 Effects:x.swap(y).

```

### 23.2.5.3 Class template stack

[stack]

- 1 Any sequence container [supporting operations back\(\), push\\_back\(\) and pop\\_back\(\)](#) that meets the requirements of [the BackInsertionContainer concept](#) can be used to instantiate stack. In particular, vector ([23.2.6](#)), list ([23.2.4](#)) and deque ([23.2.2](#)) can be used.

#### 23.2.5.3.1 stack definition

[stack.defn]

```

namespace std {
 template <classObjectType T, classStackLikeContainer Container = deque<T> >
 requires SameType<Cont::value_type, T>
 && NothrowDestructible<Cont>
 class stack {
 public:
 typedef typename Container::value_type value_type;
 typedef typename Container::reference reference;
 typedef typename Container::const_reference const_reference;
 typedef typename Container::size_type size_type;
 typedef Container container_type;
 protected:
 Container c;

 public:
 requires CopyConstructible<Cont> explicit stack(const Container&);
 requires MoveConstructible<Cont> explicit stack(Container&& = Container());
 template <class Alloc>
 requires Constructible<Cont, const Alloc&>
 explicit stack(const Alloc&);
 template <class Alloc>
 requires Constructible<Cont, const Cont&, const Alloc&>
 stack(const Container&, const Alloc&);
 template <class Alloc>
 requires Constructible<Cont, Cont&&, const Alloc&>
 stack(Container&&, const Alloc&);
 template <class Alloc>
 Constructible<Cont, Cont&&, const Alloc&>
 stack(stack&&, const Alloc&);

 bool empty() const { return c.empty(c); }
 size_type size() const { return c.size(c); }
 reference top() { return c.back(c); }
 const_reference top() const { return c.back(c); }
 void push(const value_type& x) { c.push_back(c, x); }

```

```

void push(value_type&& x) { e.push_back(c, move(x)); }
template <class... Args>
 requires BackEmplacementContainer<Cont, Args&&...>
 void emplace(Args&&... args)
 { e.emplace_back(c, forward<Args>(args)...); }
void pop() { e.pop_back(c); }
 requires Swappable<Cont>
 void swap(stack&& s) { e.swap(c, s.c); }
};

template <class EqualityComparable T, class Container>
 bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);
template <class LessThanComparable T, class Container>
 bool operator< (const stack<T, Container>& x, const stack<T, Container>& y);
template <class EqualityComparable T, class Container>
 bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);
template <class LessThanComparable T, class Container>
 bool operator> (const stack<T, Container>& x, const stack<T, Container>& y);
template <class LessThanComparable T, class Container>
 bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);
template <class LessThanComparable T, class Container>
 bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y);
template <class ObjectType T, class Swappable Container>
 void swap(stack<T, Container>& x, stack<T, Container>& y);
template <class ObjectType T, class Swappable Container>
 void swap(stack<T, Container>&& x, stack<T, Container>& y);
template <class ObjectType T, class Swappable Container>
 void swap(stack<T, Container>& x, stack<T, Container>&& y);

template <class T, class Cont, class Alloc>
 requires UsesAllocator<Cont, Alloc>
 concept_map UsesAllocator<stack<T, Cont>, Alloc> { }

template <class T, class Container, class Alloc>
 struct uses_allocator<stack<T, Container>, Alloc>
 { uses_allocator<Container, Alloc>::type { } };

template <class T, class Container>
 struct constructible_with_allocator_suffix<stack<T, Container>>
 { true_type { } };

}

```

### 23.2.5.3.2 stack operators

[stack.ops]

```

template <class EqualityComparable T, class Container>
 bool operator==(const stack<T, Container>& x,
 const stack<T, Container>& y);

```

1      *Returns:* `x.c == y.c`.

```
template <class EqualityComparable T, class Container>
 bool operator!=(const stack<T, Container>& x,
 const stack<T, Container>& y);
```

2      *Returns:*  $x.c \neq y.c$ .

```
template <class LessThanComparable T, class Container>
 bool operator< (const stack<T, Container>& x,
 const stack<T, Container>& y);
```

3      *Returns:*  $x.c < y.c$ .

```
template <class LessThanComparable T, class Container>
 bool operator<=(const stack<T, Container>& x,
 const stack<T, Container>& y);
```

4      *Returns:*  $x.c \leq y.c$ .

```
template <class LessThanComparable T, class Container>
 bool operator> (const stack<T, Container>& x,
 const stack<T, Container>& y);
```

5      *Returns:*  $x.c > y.c$ .

```
template <class LessThanComparable T, class Container>
 bool operator>=(const stack<T, Container>& x,
 const stack<T, Container>& y);
```

6      *Returns:*  $x.c \geq y.c$ .

### 23.2.5.3.3 stack specialized algorithms

[stack.special]

```
template <class ObjectType T, class Swappable Container>
 void swap(stack<T, Container>& x, stack<T, Container>& y);
template <class ObjectType T, class Swappable Container>
 void swap(stack<T, Container>&& x, stack<T, Container>& y);
template <class ObjectType T, class Swappable Container>
 void swap(stack<T, Container>& x, stack<T, Container>&& y);
```

1      *Effects:*  $x.swap(y)$ .

### 23.2.6 Class template vector

[vector]

- 1 A vector is a sequence container that supports random access iterators. In addition, it supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though hints can be given to improve efficiency. The elements of a vector are stored contiguously, meaning that if  $v$  is a `vector<T, Allocator>` where  $T$  is some type other than `bool`, then it obeys the identity  $\&v[n] == \&v[0] + n$  for all  $0 \leq n < v.size()$ .
- 2 A vector satisfies all of the requirements of a container and of a reversible container (given in two tables in 23.1), of a sequence container, including most of the optional sequence container requirements (23.1.1), and of an allocator-

aware container (Table ??). The exceptions are the `push_front` and `pop_front` member functions, which are not provided. Descriptions are provided here only for operations on vector that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
 template <class ValueType T, class Allocator Allocator = allocator<T> >
 requires MoveConstructible<T>
 class vector {
 public:
 // types:
 typedef typename Allocator::reference reference;
 typedef typename Allocator::const_reference const_reference;
 typedef implementation-defined iterator; // See 23.1
 typedef implementation-defined const_iterator; // See 23.1
 typedef implementation-defined size_type; // See 23.1
 typedef implementation-defined difference_type; // See 23.1
 typedef T value_type;
 typedef Allocator allocator_type;
 typedef typename Allocator::pointer pointer;
 typedef typename Allocator::const_pointer const_pointer;
 typedef reverse_iterator<iterator> reverse_iterator;
 typedef reverse_iterator<const_iterator> const_reverse_iterator;

 // 23.2.6.1 construct/copy/destroy:
 explicit vector(const Allocator& = Allocator());
 requires AllocatableElement<Alloc, T>
 explicit vector(size_type n);
 requires AllocatableElement<Alloc, T, const T&>
 vector(size_type n, const T& value, const Allocator& = Allocator());
 template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, T, Iter::reference>
 vector(InputIteratorIter first, InputIteratorIter last,
 const Allocator& = Allocator());
 requires AllocatableElement<Alloc, T, const T&> vector(const vector<T,Allocator>& x);
 requires AllocatableElement<Alloc, T, T&&> vector(vector&&);
 requires AllocatableElement<Alloc, T, const T&> vector(const vector&, const Allocator&);
 requires AllocatableElement<Alloc, T, T&&> vector(vector&&, const Allocator&);
 requires AllocatableElement<Alloc, T, const T&>
 vector(initializer_list<T>, const Allocator& = Allocator());
 ~vector();
 requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
 vector<T,Allocator>& operator=(const vector<T,Allocator>& x);
 requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
 vector<T,Allocator>& operator=(vector<T,Allocator>&& x);
 requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
 vector<T,Allocator>& operator=(initializer_list<T>);
 template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, T, Iter::reference>
 && HasAssign<T, Iter::reference>
 void assign(InputIteratorIter first, InputIteratorIter last);
```

```

requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
 void assign(size_type n, const T& u);
requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
 void assign(initializer_list<T>);
 allocator_type get_allocator() const;

// iterators:
 iterator begin();
 const_iterator begin() const;
 iterator end();
 const_iterator end() const;
 reverse_iterator rbegin();
 const_reverse_iterator rbegin() const;
 reverse_iterator rend();
 const_reverse_iterator rend() const;

 const_iterator cbegin() const;
 const_iterator cend() const;
 const_reverse_iterator crbegin() const;
 const_reverse_iterator crend() const;

// 23.2.6.2 capacity:
 size_type size() const;
 size_type max_size() const;
requires AllocatableElement<Alloc, T>
 void resize(size_type sz);
requires AllocatableElement<Alloc, T, const T&>
 void resize(size_type sz, const T& c);
 size_type capacity() const;
 bool empty() const;
 void reserve(size_type n);
 void shrink_to_fit();

// element access:
 reference operator[](size_type n);
 const_reference operator[](size_type n) const;
 const_reference at(size_type n) const;
 reference at(size_type n);
 reference front();
 const_reference front() const;
 reference back();
 const_reference back() const;

// 23.2.6.3 data access
 pointer data();
 const_pointer data() const;

// 23.2.6.4 modifiers:
 template <class... Args>
 requires AllocatableElement<Alloc, T, Args&&...>

```

```

 void emplace_back(Args&&... args);
 requires AllocatableElement<Alloc, T, const T&> void push_back(const T& x);
 requires AllocatableElement<Alloc, T, T&&> void push_back(T&& x);
 void pop_back();

template <class... Args>
 requires AllocatableElement<Alloc, T, Args&&...>
 && MoveAssignable<T>
 iterator emplace(const_iterator position, Args&&... args);
 requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
 iterator insert(const_iterator position, const T& x);
 requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
 void insert(const_iterator position, T&& x);
 requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
 void insert(const_iterator position, size_type n, const T& x);
template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, T, Iter::reference>
 && MoveAssignable<T>
 void insert(const_iterator position,
 InputIteratorIter first, InputIteratorIter last);
 requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
 void insert(const_iterator position, initializer_list<T> il);
 requires MoveAssignable<T> iterator erase(const_iterator position);
 requires MoveAssignable<T> iterator erase(const_iterator first, const_iterator last);
 void swap(vector<T, Allocator>&&);
 void clear();
};

template <class EqualityComparable T, class Allocator>
 bool operator==(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
template <class LessThanComparable T, class Allocator>
 bool operator< (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
template <class EqualityComparable T, class Allocator>
 bool operator!=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
template <class LessThanComparable T, class Allocator>
 bool operator> (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
template <class LessThanComparable T, class Allocator>
 bool operator>=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
template <class LessThanComparable T, class Allocator>
 bool operator<=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);

// specialized algorithms:
template <class ValueType T, class Allocator>
 void swap(vector<T, Allocator>& x, vector<T, Allocator>& y);
template <class ValueType T, class Allocator>
 void swap(vector<T, Allocator>&& x, vector<T, Allocator>& y);
template <class ValueType T, class Allocator>
 void swap(vector<T, Allocator>& x, vector<T, Allocator>&& y);

template <class T, class Alloc

```

```

 struct constructible_with_allocator_suffix<vector<T, Alloc>>
 : true_type { };
 }

```

### 23.2.6.1 vector constructors, copy, and assignment

[vector.cons]

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

1 *Effects*: Constructs an empty vector, using the specified allocator.

2 *Complexity*: Constant.

```
requires AllocatableElement<Alloc, T> explicit vector(size_type n);
```

3 *Effects*: Constructs a vector with  $n$  default constructed elements.

4 ~~*Requires*:  $T$  shall be DefaultConstructible.~~

5 *Complexity*: Linear in  $n$ .

```
requires AllocatableElement<Alloc, T, const T&>
explicit vector(size_type n, const T& value,
 const Allocator& = Allocator());
```

6 *Effects*: Constructs a vector with  $n$  copies of `value`, using the specified allocator.

7 ~~*Requires*:  $T$  shall be CopyConstructible.~~

8 *Complexity*: Linear in  $n$ .

```
template <class InputIterator InputIterator Iter>
requires AllocatableElement<Alloc, T, Iter::reference>
vector(InputIterator first, InputIterator last,
 const Allocator& = Allocator());
```

9 *Effects*: Constructs a vector equal to the range `[first, last)`, using the specified allocator.

10 *Complexity*: Makes only  $N$  calls to the copy constructor of  $T$  (where  $N$  is the distance between `first` and `last`) and no reallocations if ~~iterators `first` and `last` are of forward, bidirectional, or random access categories~~ `Iter` meets the requirements of the ForwardIterator concept. It makes order  $N$  calls to the copy constructor of  $T$  and order  $\log(N)$  reallocations if they are just input iterators.

```
template <class InputIterator InputIterator Iter>
requires AllocatableElement<Alloc, T, Iter::reference>
 && HasAssign<T, Iter::reference>
void assign(InputIterator first, InputIterator last);
```

11 *Effects*:

```
 erase(begin(), end());
 insert(begin(), first, last);
```

```
requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
void assign(size_type n, const T& t);
```

12 *Effects:*

```
erase(begin(), end());
insert(begin(), n, t);
```

### 23.2.6.2 vector capacity

[vector.capacity]

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

1 *Returns:* The total number of elements that the vector can hold without requiring reallocation.

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

2 *Requires:* If value\_type has a move constructor, that constructor shall not throw any exceptions.

3 *Effects:* A directive that informs a vector of a planned change in size, so that it can manage the storage allocation accordingly. After reserve(), capacity() is greater or equal to the argument of reserve if reallocation happens; and equal to the previous value of capacity() otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of reserve(). If an exception is thrown, there are no effects.

4 *Complexity:* It does not change the size of the sequence and takes at most linear time in the size of the sequence.

5 *Throws:* length\_error if  $n > \text{max\_size}()$ .<sup>3)</sup>

6 *Remarks:* Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during insertions that happen after a call to reserve() until the time when an insertion would make the size of the vector greater than the value of capacity().

```
void shrink_to_fit();
```

7 *Remarks:* shrink\_to\_fit is a non-binding request to reduce capacity() to size(). [Note: The request is non-binding to allow latitude for implementation-specific optimizations. — end note]

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

8 *Effects:* Exchanges the contents and capacity() of \*this with that of x.

9 *Complexity:* Constant time.

requires AllocatableElement<Alloc, T>

```
void resize(size_type sz);
```

10 *Effects:* If  $sz < \text{size}()$ , equivalent to `erase(begin() + sz, end())`; If  $\text{size}() < sz$ , appends  $sz - \text{size}()$  default constructed elements to the sequence.

11 ~~*Requires:* T shall be DefaultConstructible.~~

requires AllocatableElement<Alloc, T, const T&>

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

12 *Effects:*

---

<sup>3)</sup> reserve() uses Allocator::allocate() which may throw an appropriate exception.

```

 if (sz > size())
 insert(end(), sz-size(), c);
 else if (sz < size())
 erase(begin()+sz, end());
 else
 ; // do nothing

```

13 *Requires:* If `value_type` has a move constructor, that constructor shall not throw any exceptions.

### 23.2.6.3 vector data

[vector.data]

```

pointer data();
const_pointer data() const;

```

1 *Returns:* A pointer such that `[data(), data() + size())` is a valid range. For a non-empty vector, `data() == &front()`.

2 *Complexity:* Constant time.

3 *Throws:* Nothing.

### 23.2.6.4 vector modifiers

[vector.modifiers]

```

requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
 iterator insert(const_iterator position, const T& x);
requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
 iterator insert(const_iterator position, T&& x);
requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
 void insert(const_iterator position, size_type n, const T& x);
template <class InputIterator, InputIterator Iter>
 requires AllocatableElement<Alloc, T, Iter::reference>
 && MoveAssignable<T>
 void insert(const_iterator position,
 InputIterator first, InputIterator last);

template <class... Args>
 requires AllocatableElement<Alloc, T, Args&&...>
 void emplace_back(Args&&... args);
template <class... Args>
 requires AllocatableElement<Alloc, T, Args&&...>
 && MoveAssignable<T>
 iterator emplace(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, T, const T&> void push_back(const T& x);
requires AllocatableElement<Alloc, T, T&&> void push_back(T&& x);

```

1 *Requires:* If `value_type` has a move constructor, that constructor shall not throw any exceptions.

2 *Remarks:* Causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid. If an exception is thrown other than by the copy

constructor or assignment operator of T or by any InputIterator operation there are no effects.

- 3 *Complexity:* The complexity is linear in the number of elements inserted plus the distance to the end of the vector.

```
requires MoveAssignable<T> iterator erase(const_iterator position);
requires MoveAssignable<T> iterator erase(const_iterator first, const_iterator last);
```

- 4 *Effects:* Invalidates iterators and references at or after the point of the erase.

- 5 *Complexity:* The destructor of T is called the number of times equal to the number of the elements erased, but the move assignment operator of T is called the number of times equal to the number of elements in the vector after the erased elements.

- 6 *Throws:* Nothing unless an exception is thrown by the copy constructor or assignment operator of T.

### 23.2.6.5 vector specialized algorithms

[vector.special]

```
template <classValueType T, class Allocator>
 void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);
template <classValueType T, class Allocator>
 void swap(vector<T,Allocator>&& x, vector<T,Allocator>& y);
template <classValueType T, class Allocator>
 void swap(vector<T,Allocator>& x, vector<T,Allocator>&& y);
```

- 1 *Effects:*

```
x.swap(y);
```

### 23.2.7 Class vector<bool>

[vector.bool]

- 1 To optimize space allocation, a specialization of vector for bool elements is provided:

```
namespace std {
 template <classAllocator Allocator> class vector<bool, Allocator> {
 public:
 // types:
 typedef bool const_reference;
 typedef implementation-defined iterator; // See 23.1
 typedef implementation-defined const_iterator; // See 23.1
 typedef implementation-defined size_type; // See 23.1
 typedef implementation-defined difference_type; // See 23.1
 typedef bool value_type;
 typedef Allocator allocator_type;
 typedef implementation-defined pointer;
 typedef implementation-defined const_pointer;
 typedef reverse_iterator<iterator> reverse_iterator;
 typedef reverse_iterator<const_iterator> const_reverse_iterator;

 // bit reference:
 class reference {
```

```

 friend class vector;
 reference();
public:
 ~reference();
 operator bool() const;
 reference& operator=(const bool x);
 reference& operator=(const reference& x);
 void flip(); // flips the bit
};

// construct/copy/destroy:
explicit vector(const Allocator& = Allocator());
explicit vector(size_type n, const bool& value = bool(),
 const Allocator& = Allocator());
template <class InputIterator, InputIterator Iter>
 requires Convertible<Iter::reference, bool>
 vector(InputIterator first, InputIterator last,
 const Allocator& = Allocator());
vector(const vector<bool, Allocator>& x);
vector(vector<bool, Allocator>&& x);
vector(const vector&, const Allocator&);
vector(vector&&, const Allocator&);
vector(initializer_list<bool>);
~vector();
vector<bool, Allocator>& operator=(const vector<bool, Allocator>& x);
vector<bool, Allocator>& operator=(vector<bool, Allocator>&& x);
vector<bool, Allocator>& operator=(initializer_list<bool>);
template <class InputIterator, InputIterator Iter>
 requires Convertible<Iter::reference, bool>
 void assign(InputIterator first, InputIterator last);
void assign(size_type n, const bool& t);
void assign(initializer_list<bool>);
allocator_type get_allocator() const;

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:

```

```

size_type size() const;
size_type max_size() const;
void resize(size_type sz, bool c = false);
size_type capacity() const;
bool empty() const;
void reserve(size_type n);
void shrink_to_fit();

// element access:
reference operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference at(size_type n);
reference front();
const_reference front() const;
reference back();
const_reference back() const;

// modifiers:
void push_back(const bool& x);
void pop_back();
iterator insert(const_iterator position, const bool& x);
void insert (const_iterator position, size_type n, const bool& x);
template <class InputIterator InputIterator Iter>
 requires Convertible<Iter::reference, bool>
 void insert(const_iterator position,
 InputIteratorIter first, InputIteratorIter last);
void insert(const_iterator position, initializer_list<bool> il);

iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void swap(vector<bool, Allocator>&&);
static void swap(reference x, reference y);
void flip(); // flips all bits
void clear();
};

}

```

- 2 Unless described below, all operations have the same requirements and semantics as the primary `vector` template, except that operations dealing with the `bool` value type map to bit values in the container storage.
- 3 There is no requirement that the data be stored as a contiguous allocation of `bool` values. A space-optimized representation of bits is recommended instead.
- 4 `reference` is a class that simulates the behavior of references of a single bit in `vector<bool>`. The conversion operator returns `true` when the bit is set, and `false` otherwise. The assignment operator sets the bit when the argument is (convertible to) `true` and clears it otherwise. `flip` reverses the state of the bit.

```
void flip();
```

- 5 *Effects*: Replaces each element in the container with its complement. It is unspecified whether the function has any effect on allocated but unused bits.

### 23.3 Associative containers

[associative]

- 1 Headers `<map>` and `<set>`:

#### Header `<map>` synopsis

```
namespace std {
 template <classValueType Key, classValueType T,
 classPredicate<auto, Key, Key> Compare = less<Key>,
 classAllocator Allocator = allocator<pair<const Key, T> > >
 requires NothrowDestructible<Key> && NothrowDestructible<T> && CopyConstructible<Compare>
 && AllocatableElement<Alloc, Compare, const Compare&>
 && AllocatableElement<Alloc, Compare, Compare&&>
 class map;
 template <classEqualityComparable Key, classEqualityComparable T, class Compare, class Allocator>
 bool operator==(const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);
 template <classLessThanComparable Key, classLessThanComparable T, class Compare, class Allocator>
 bool operator< (const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);
 template <classEqualityComparable Key, classEqualityComparable T, class Compare, class Allocator>
 bool operator!=(const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);
 template <classLessThanComparable Key, classLessThanComparable T, class Compare, class Allocator>
 bool operator> (const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);
 template <classLessThanComparable Key, classLessThanComparable T, class Compare, class Allocator>
 bool operator>=(const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);
 template <classLessThanComparable Key, classLessThanComparable T, class Compare, class Allocator>
 bool operator<=(const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);
 template <classValueType Key, classValueType T, class Compare, class Allocator>
 void swap(map<Key,T,Compare,Allocator>& x,
 map<Key,T,Compare,Allocator>& y);
 template <classValueType Key, classValueType T, class Compare, class Allocator>
 void swap(map<Key,T,Compare,Allocator&& x,
 map<Key,T,Compare,Allocator>& y);
 template <classValueType Key, classValueType T, class Compare, class Allocator>
 void swap(map<Key,T,Compare,Allocator& x,
 map<Key,T,Compare,Allocator&& y);

 template <classValueType Key, classValueType T,
 classPredicate<auto, Key, Key> Compare = less<Key>,
 classAllocator Allocator = allocator<pair<const Key, T> > >
 requires NothrowDestructible<Key> && NothrowDestructible<T> && CopyConstructible<Compare>
 && AllocatableElement<Alloc, Compare, const Compare&>
```

```

 && AllocatableElement<Alloc, Compare, Compare&&>
 class multimap;
 template <class EqualityComparable Key, class EqualityComparable T, class Compare, class Allocator>
 bool operator==(const multimap<Key,T,Compare,Allocator>& x,
 const multimap<Key,T,Compare,Allocator>& y);
 template <class LessThanComparable Key, class LessThanComparable T, class Compare, class Allocator>
 bool operator< (const multimap<Key,T,Compare,Allocator>& x,
 const multimap<Key,T,Compare,Allocator>& y);
 template <class EqualityComparable Key, class EqualityComparable T, class Compare, class Allocator>
 bool operator!=(const multimap<Key,T,Compare,Allocator>& x,
 const multimap<Key,T,Compare,Allocator>& y);
 template <class LessThanComparable Key, class LessThanComparable T, class Compare, class Allocator>
 bool operator> (const multimap<Key,T,Compare,Allocator>& x,
 const multimap<Key,T,Compare,Allocator>& y);
 template <class LessThanComparable Key, class LessThanComparable T, class Compare, class Allocator>
 bool operator>= (const multimap<Key,T,Compare,Allocator>& x,
 const multimap<Key,T,Compare,Allocator>& y);
 template <class LessThanComparable Key, class LessThanComparable T, class Compare, class Allocator>
 bool operator<= (const multimap<Key,T,Compare,Allocator>& x,
 const multimap<Key,T,Compare,Allocator>& y);
 template <class ValueType Key, class ValueType T, class Compare, class Allocator>
 void swap(multimap<Key,T,Compare,Allocator>& x,
 multimap<Key,T,Compare,Allocator>& y);
 template <class ValueType Key, class ValueType T, class Compare, class Allocator>
 void swap(multimap<Key,T,Compare,Allocator>&& x,
 multimap<Key,T,Compare,Allocator>& y);
 template <class ValueType Key, class ValueType T, class Compare, class Allocator>
 void swap(multimap<Key,T,Compare,Allocator>& x,
 multimap<Key,T,Compare,Allocator>&& y);
}

```

### Header <set> synopsis

```

namespace std {
 template <class ValueType Key, class Predicate<auto, Key, Key> Compare = less<Key>,
 class Allocator Allocator = allocator<Key> >
 requires NothrowDestructible<Key> && CopyConstructible<Compare>
 && AllocatableElement<Alloc, Compare, const Compare&>
 && AllocatableElement<Alloc, Compare, Compare&&>
 class set;
 template <class EqualityComparable Key, class Compare, class Allocator>
 bool operator==(const set<Key,Compare,Allocator>& x,
 const set<Key,Compare,Allocator>& y);
 template <class LessThanComparable Key, class Compare, class Allocator>
 bool operator< (const set<Key,Compare,Allocator>& x,
 const set<Key,Compare,Allocator>& y);
 template <class EqualityComparable Key, class Compare, class Allocator>
 bool operator!=(const set<Key,Compare,Allocator>& x,
 const set<Key,Compare,Allocator>& y);
 template <class LessThanComparable Key, class Compare, class Allocator>

```

```

 bool operator> (const set<Key,Compare,Allocator>& x,
 const set<Key,Compare,Allocator>& y);
template <classLessThanComparable Key, class Compare, class Allocator>
 bool operator>=(const set<Key,Compare,Allocator>& x,
 const set<Key,Compare,Allocator>& y);
template <classLessThanComparable Key, class Compare, class Allocator>
 bool operator<=(const set<Key,Compare,Allocator>& x,
 const set<Key,Compare,Allocator>& y);
template <classValueType Key, class Compare, class Allocator>
 void swap(set<Key,Compare,Allocator>& x,
 set<Key,Compare,Allocator>& y);
template <classValueType Key, class T, class Compare, class Allocator>
 void swap(set<Key,T,Compare,Allocator>&& x,
 set<Key,T,Compare,Allocator>& y);
template <classValueType Key, class T, class Compare, class Allocator>
 void swap(set<Key,T,Compare,Allocator>& x,
 set<Key,T,Compare,Allocator>&& y);

template <classValueType Key, classPredicate<auto, Key, Key> Compare = less<Key>,
 classAllocator Allocator = allocator<Key> >
 requires NothrowDestructible<Key> && CopyConstructible<Compare>
 && AllocatableElement<Alloc, Compare, const Compare>
 && AllocatableElement<Alloc, Compare, Compare>&&
 class multiset;
template <classEqualityComparable Key, class Compare, class Allocator>
 bool operator==(const multiset<Key,Compare,Allocator>& x,
 const multiset<Key,Compare,Allocator>& y);
template <classLessThanComparable Key, class Compare, class Allocator>
 bool operator< (const multiset<Key,Compare,Allocator>& x,
 const multiset<Key,Compare,Allocator>& y);
template <classEqualityComparable Key, class Compare, class Allocator>
 bool operator!=(const multiset<Key,Compare,Allocator>& x,
 const multiset<Key,Compare,Allocator>& y);
template <classLessThanComparable Key, class Compare, class Allocator>
 bool operator> (const multiset<Key,Compare,Allocator>& x,
 const multiset<Key,Compare,Allocator>& y);
template <classLessThanComparable Key, class Compare, class Allocator>
 bool operator>=(const multiset<Key,Compare,Allocator>& x,
 const multiset<Key,Compare,Allocator>& y);
template <classLessThanComparable Key, class Compare, class Allocator>
 bool operator<=(const multiset<Key,Compare,Allocator>& x,
 const multiset<Key,Compare,Allocator>& y);
template <classValueType Key, class Compare, class Allocator>
 void swap(multiset<Key,Compare,Allocator>& x,
 multiset<Key,Compare,Allocator>& y);
template <classValueType Key, class T, class Compare, class Allocator>
 void swap(multiset<Key,T,Compare,Allocator>&& x,
 multiset<Key,T,Compare,Allocator>& y);
template <classValueType Key, class T, class Compare, class Allocator>
 void swap(multiset<Key,T,Compare,Allocator>& x,
 multiset<Key,T,Compare,Allocator>&& y);

```

```

 multiset<Key,T,Compare,Allocator>&& y);
}

```

### 23.3.1 Class template map

[map]

- 1 A map is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type T based on the keys. The map class supports bidirectional iterators.
- 2 A map satisfies all of the requirements of a container, of a reversible container (23.1), of an associative container (23.1.2), and of an allocator-aware container (Table ??). A map also provides most operations described in (23.1.2) for unique keys. This means that a map supports the a\_uniq operations in (23.1.2) but not the a\_eq operations. For a map<Key,T> the key\_type is Key and the value\_type is pair<const Key,T>. Descriptions are provided here only for operations on map that are not described in one of those tables or for operations where there is additional semantic information.

```

namespace std {
 template <class ValueType Key, class ValueType T,
 class Predicate<auto, Key, Key> Compare = less<Key>,
 class Allocator Allocator = allocator<pair<const Key, T> > >
 requires NothrowDestructible<Key> && NothrowDestructible<T> && CopyConstructible<Compare>
 && AllocatableElement<Alloc, Compare, const Compare&>
 && AllocatableElement<Alloc, Compare, Compare&&>
 class map {
 public:
 // types:
 typedef Key key_type;
 typedef T mapped_type;
 typedef pair<const Key, T> value_type;
 typedef Compare key_compare;
 typedef Allocator allocator_type;
 typedef typename Allocator::reference reference;
 typedef typename Allocator::const_reference const_reference;
 typedef implementation-defined iterator; // See 23.1
 typedef implementation-defined const_iterator; // See 23.1
 typedef implementation-defined size_type; // See 23.1
 typedef implementation-defined difference_type; // See 23.1
 typedef typename Allocator::pointer pointer;
 typedef typename Allocator::const_pointer const_pointer;
 typedef reverse_iterator<iterator> reverse_iterator;
 typedef reverse_iterator<const_iterator> const_reverse_iterator;

 class value_compare
 : public binary_function<value_type,value_type,bool> {
 friend class map;
 protected:
 Compare comp;
 value_compare(Compare c) : comp(c) {}
 public:
 bool operator()(const value_type& x, const value_type& y) const {
 return comp(x.first, y.first);
 }
 }
 }
}

```

```

};

// 23.3.1.1 construct/copy/destroy:
explicit map(const Compare& comp = Compare(),
 const Allocator& = Allocator());
template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 map(InputIteratorIter first, InputIteratorIter last,
 const Compare& comp = Compare(), const Allocator& = Allocator());
 requires AllocatableElement<Alloc, value_type, const value_type&>
 map(const map<Key,T,Compare,Allocator>& x);
 map(map<Key,T,Compare,Allocator>&& x);
 map(const Allocator&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 map(const map&, const Allocator&);
 requires AllocatableElement<Alloc, value_type, value_type&&>
 map(map&&, const Allocator&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 map(initializer_list<value_type>,
 const Compare& = Compare(),
 const Allocator& = Allocator());
~map();
 requires AllocatableElement<Alloc, value_type, const value_type&>
 && CopyAssignable<value_type>
 map<Key,T,Compare,Allocator>& operator=(const map<Key,T,Compare,Allocator>& x);
 map<Key,T,Compare,Allocator>&
 operator=(map<Key,T,Compare,Allocator>&& x);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 && CopyAssignable<value_type>
 map<Key,T,Compare,Allocator>& operator=(initializer_list<value_type>);
 allocator_type get_allocator() const;

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:

```

```

bool empty() const;
size_type size() const;
size_type max_size() const;

// 23.3.1.2 element access:
requires AllocatableElement<Alloc, value_type, const key_type&, mapped_type&&>
 && AllocatableElement<Alloc, mapped_type>
 T& operator[](const key_type& x);
requires AllocatableElement<Alloc, value_type, key_type&&, mapped_type&&>
 && AllocatableElement<Alloc, mapped_type>
 T& operator[](key_type&& x);
T& at(const key_type& x);
const T& at(const key_type& x) const;

// modifiers:
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 iterator emplace_hint(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, value_type, const value_type&>
 pair<iterator, bool> insert(const value_type& x);
template <class P>
 requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
 pair<iterator, bool> insert(P&& x);
requires AllocatableElement<Alloc, value_type, const value_type&>
 iterator insert(const_iterator position, const value_type& x);
template <class P>
 requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
 iterator insert(const_iterator position, P&&);
template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 void insert(InputIteratorIter first, InputIteratorIter last);
requires AllocatableElement<Alloc, value_type, const value_type&>
 void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(map<Key,T,Compare,Allocater>&&);
void clear();

// observers:
key_compare key_comp() const;
value_compare value_comp() const;

// 23.3.1.4 map operations:
iterator find(const key_type& x);

```

```

const_iterator find(const key_type& x) const;
size_type count(const key_type& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;

pair<iterator,iterator>
 equal_range(const key_type& x);
pair<const_iterator,const_iterator>
 equal_range(const key_type& x) const;
};

template <class EqualityComparable Key, class EqualityComparable T, class Compare, class Allocator>
 bool operator==(const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);
template <class LessThanComparable Key, class LessThanComparable T, class Compare, class Allocator>
 bool operator< (const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);
template <class EqualityComparable Key, class EqualityComparable T, class Compare, class Allocator>
 bool operator!=(const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);
template <class LessThanComparable Key, class LessThanComparable T, class Compare, class Allocator>
 bool operator> (const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);
template <class LessThanComparable Key, class LessThanComparable T, class Compare, class Allocator>
 bool operator>=(const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);
template <class LessThanComparable Key, class LessThanComparable T, class Compare, class Allocator>
 bool operator<=(const map<Key,T,Compare,Allocator>& x,
 const map<Key,T,Compare,Allocator>& y);

// specialized algorithms:
template <class ValueType Key, class ValueType T, class Compare, class Allocator>
 void swap(map<Key,T,Compare,Allocator>& x,
 map<Key,T,Compare,Allocator>& y);
template <class ValueType Key, class ValueType T, class Compare, class Allocator>
 void swap(map<Key,T,Compare,Allocator&& x,
 map<Key,T,Compare,Allocator>& y);
template <class ValueType Key, class ValueType T, class Compare, class Allocator>
 void swap(map<Key,T,Compare,Allocator& x,
 map<Key,T,Compare,Allocator&& y);

template <class Key, class T, class Compare, class Alloc>
 struct constructible_with_allocator_suffix<
 map<Key, T, Compare, Alloc>->
 :-true_type{-};
}

```

## 23.3.1.1 map constructors, copy, and assignment

[map.cons]

```
explicit map(const Compare& comp = Compare(),
 const Allocator& = Allocator());
```

1 *Effects:* Constructs an empty map using the specified comparison object and allocator.

2 *Complexity:* Constant.

```
template <class InputIterator, InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
map(InputIterator first, InputIterator last,
 const Compare& comp = Compare(), const Allocator& = Allocator());
```

3 *Requires:* If the iterator's dereference operator returns an lvalue or a const rvalue pair<key\_type, mapped\_type>, then both key\_type and mapped\_type shall be CopyConstructible.

4 *Effects:* Constructs an empty map using the specified comparison object and allocator, and inserts elements from the range  $[first, last)$ .

5 *Complexity:* Linear in  $N$  if the range  $[first, last)$  is already sorted using *comp* and otherwise  $N \log N$ , where  $N$  is  $last - first$ .

## 23.3.1.2 map element access

[map.access]

```
requires AllocatableElement<Alloc, value_type, const key_type&>, mapped_type&&>
 && AllocatableElement<Alloc, mapped_type>
T& operator[](const key_type& x);
```

1 *Effects:* If there is no key equivalent to  $x$  in the map, inserts  $value\_type(x, T())$  into the map.

2 *Requires:* key\_type shall be CopyConstructible and mapped\_type shall be DefaultConstructible.

3 *Returns:* A reference to the mapped\_type corresponding to  $x$  in  $*this$ .

4 *Complexity:* logarithmic.

```
requires AllocatableElement<Alloc, value_type, key_type&&, mapped_type&&>
 && AllocatableElement<Alloc, mapped_type>
T& operator[](key_type&& x);
```

5 *Effects:* If there is no key equivalent to  $x$  in the map, inserts  $value\_type(move(x), T())$  into the map.

6 *Requires:* mapped\_type shall be DefaultConstructible.

7 *Returns:* A reference to the mapped\_type corresponding to  $x$  in  $*this$ .

8 *Complexity:* logarithmic.

```
T& at(const key_type& x);
const T& at(const key_type& x) const;
```

- 9 *Returns:* A reference to the element whose key is equivalent to `x`.
- 10 *Throws:* An exception object of type `out_of_range` if no such element is present.
- 11 *Complexity:* logarithmic.

### 23.3.1.3 map modifiers

[map.modifiers]

```
template <class P>
 requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
 pair<iterator, bool> insert(P&& x);
template <class P>
 requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
 pair<iterator, bool> insert(const_iterator position, P&& x);
```

- 1 *Requires:* ~~P shall be convertible to value\_type.~~

If `P` is instantiated as a reference type, then the argument `x` is copied from. Otherwise `x` is considered to be an rvalue as it is converted to `value_type` and inserted into the map. Specifically, in such cases `CopyConstructible` is not required of `key_type` or `mapped_type` unless the conversion from `P` specifically requires it (e.g. if `P` is a `tuple<const key_type, mapped_type>`, then `key_type` must be `CopyConstructible`). The signature taking `InputIterator` parameters does not require `CopyConstructible` of either `key_type` or `mapped_type` if the dereferenced `InputIterator` returns a non-const rvalue `pair<key_type, mapped_type>`. Otherwise `CopyConstructible` is required for both `key_type` and `mapped_type`.

### 23.3.1.4 map operations

[map.ops]

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

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type &x) const;

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

- 1 The `find`, `lower_bound`, `upper_bound` and `equal_range` member functions each have two versions, one `const` and the other non-`const`. In each case the behavior of the two functions is identical except that the `const` version returns a `const_iterator` and the non-`const` version an `iterator` (23.1.2).

### 23.3.1.5 map specialized algorithms

[map.special]

```

template <classValueType Key, classValueType T, class Compare, class Allocator>
 void swap(map<Key,T,Compare,Allocator>& x,
 map<Key,T,Compare,Allocator>& y);
template <classValueType Key, classValueType T, class Compare, class Allocator>
 void swap(map<Key,T,Compare,Allocator>&& x,
 map<Key,T,Compare,Allocator>& y);
template <classValueType Key, classValueType T, class Compare, class Allocator>
 void swap(map<Key,T,Compare,Allocator>& x,
 map<Key,T,Compare,Allocator>&& y);

```

1 *Effects:*

```
x.swap(y);
```

### 23.3.2 Class template multimap

[multimap]

- 1 A multimap is an associative container that supports equivalent keys (possibly containing multiple copies of the same key value) and provides for fast retrieval of values of another type T based on the keys. The multimap class supports bidirectional iterators.
- 2 A multimap satisfies all of the requirements of a container and of a reversible container (23.1), of an associative container (23.1.2), and of an allocator-aware container (Table ??). A multimap also provides most operations described in (23.1.2) for equal keys. This means that a multimap supports the `a_eq` operations in (23.1.2) but not the `a_uniq` operations. For a `multimap<Key,T>` the `key_type` is `Key` and the `value_type` is `pair<const Key,T>`. Descriptions are provided here only for operations on multimap that are not described in one of those tables or for operations where there is additional semantic information.

```

namespace std {
 template <classValueType Key, classValueType T,
 classPredicate<auto, Key, Key> Compare = less<Key>,
 classAllocator Allocator = allocator<pair<const Key, T> > >
 requires NothrowDestructible<Key> && NothrowDestructible<T> && CopyConstructible<Compare>
 && AllocatableElement<Alloc, Compare, const Compare&>
 && AllocatableElement<Alloc, Compare, Compare&&>
 class multimap {
 public:
 // types:
 typedef Key key_type;
 typedef T mapped_type;
 typedef pair<const Key,T> value_type;
 typedef Compare key_compare;
 typedef Allocator allocator_type;
 typedef typename Allocator::reference reference;
 typedef typename Allocator::const_reference const_reference;
 typedef implementation-defined iterator; // See 23.1
 typedef implementation-defined const_iterator; // See 23.1
 typedef implementation-defined size_type; // See 23.1
 typedef implementation-defined difference_type; // See 23.1
 typedef typename Allocator::pointer pointer;
 typedef typename Allocator::const_pointer const_pointer;

```

```

typedef reverse_iterator<iterator> reverse_iterator;
typedef reverse_iterator<const_iterator> const_reverse_iterator;

class value_compare
: public binary_function<value_type,value_type,bool> {
friend class multimap;
protected:
 Compare comp;
 value_compare(Compare c) : comp(c) { }
public:
 bool operator()(const value_type& x, const value_type& y) const {
 return comp(x.first, y.first);
 }
};

// construct/copy/destroy:
explicit multimap(const Compare& comp = Compare(),
 const Allocator& = Allocator());
template <class InputIterator, InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 multimap(InputIterator first, InputIterator last,
 const Compare& comp = Compare(), const Allocator& = Allocator());
 requires AllocatableElement<Alloc, value_type, const value_type&>
 multimap(const multimap<Key,T,Compare,Allocator>& x);
 multimap(multimap<Key,T,Compare,Allocator>&& x);
 multimap(const Allocator&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 multimap(const multimap&, const Allocator&);
 requires AllocatableElement<Alloc, value_type, value_type&&>
 multimap(multimap&&, const Allocator&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 multimap(initializer_list<value_type>,
 const Compare& = Compare(),
 const Allocator& = Allocator());
~multimap();
 requires AllocatableElement<Alloc, value_type, const value_type&>
 && CopyAssignable<value_type>
 multimap<Key,T,Compare,Allocator>& operator=(const multimap<Key,T,Compare,Allocator>& x);
 multimap<Key,T,Compare,Allocator>&
 operator=(const multimap<Key,T,Compare,Allocator>&& x);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 && CopyAssignable<value_type>
 multimap<Key,T,Compare,Allocator>& operator=(initializer_list<value_type>);
 allocator_type get_allocator() const;

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();

```

```

const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
bool empty() const;
size_type size() const;
size_type max_size() const;

// modifiers:
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 iterator emplace(Args&&... args);
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 iterator emplace_hint(const_iterator position, Args&&... args);
template <class... Args>
 requires AllocatableElement<Alloc, value_type, const value_type&>
 iterator insert(const value_type& x);
template <class P>
 requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
 iterator insert(P&& x);
template <class P>
 requires AllocatableElement<Alloc, value_type, const value_type&>
 iterator insert(const_iterator position, const value_type& x);
template <class P>
 requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
 iterator insert(const_iterator position, P&& x);
template <class InputIterator>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 void insert(InputIterator first, InputIterator last);
template <class InputIterator>
 requires AllocatableElement<Alloc, value_type, const value_type&>
 void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(multimap<Key,T,Compare,Allocator>&&);
void clear();

// observers:
key_compare key_comp() const;
value_compare value_comp() const;

```

```

// map operations:
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
size_type count(const key_type& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;

pair<iterator,iterator>
 equal_range(const key_type& x);
pair<const_iterator,const_iterator>
 equal_range(const key_type& x) const;
};

template <class EqualityComparable Key, class EqualityComparable T, class Compare, class Allocator>
 bool operator==(const multimap<Key,T,Compare,Allocator>& x,
 const multimap<Key,T,Compare,Allocator>& y);
template <class LessThanComparable Key, class LessThanComparable T, class Compare, class Allocator>
 bool operator< (const multimap<Key,T,Compare,Allocator>& x,
 const multimap<Key,T,Compare,Allocator>& y);
template <class EqualityComparable Key, class EqualityComparable T, class Compare, class Allocator>
 bool operator!=(const multimap<Key,T,Compare,Allocator>& x,
 const multimap<Key,T,Compare,Allocator>& y);
template <class LessThanComparable Key, class LessThanComparable T, class Compare, class Allocator>
 bool operator> (const multimap<Key,T,Compare,Allocator>& x,
 const multimap<Key,T,Compare,Allocator>& y);
template <class LessThanComparable Key, class LessThanComparable T, class Compare, class Allocator>
 bool operator>=(const multimap<Key,T,Compare,Allocator>& x,
 const multimap<Key,T,Compare,Allocator>& y);
template <class LessThanComparable Key, class LessThanComparable T, class Compare, class Allocator>
 bool operator<=(const multimap<Key,T,Compare,Allocator>& x,
 const multimap<Key,T,Compare,Allocator>& y);

// specialized algorithms:
template <class ValueType Key, class ValueType T, class Compare, class Allocator>
 void swap(multimap<Key,T,Compare,Allocator>& x,
 multimap<Key,T,Compare,Allocator>& y);
template <class ValueType Key, class ValueType T, class Compare, class Allocator>
 void swap(multimap<Key,T,Compare,Allocator>&& x,
 multimap<Key,T,Compare,Allocator>& y);
template <class ValueType Key, class ValueType T, class Compare, class Allocator>
 void swap(multimap<Key,T,Compare,Allocator>& x,
 multimap<Key,T,Compare,Allocator>&& y);

template <class Key, class T, class Compare, class Alloc>
 struct constructible_with_allocator_suffix<
 multimap<Key, T, Compare, Alloc>->

```

```

 ÷ true_type { };
}

```

### 23.3.2.1 multimap constructors

[multimap.cons]

```

explicit multimap(const Compare& comp = Compare(),
 const Allocator& = Allocator());

```

1 *Effects:* Constructs an empty multimap using the specified comparison object and allocator.

2 *Complexity:* Constant.

```

template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
multimap(InputIteratorIter first, InputIteratorIter last,
 const Compare& comp = Compare(), const Allocator& = Allocator());

```

3 *Requires:* If the iterator's dereference operator returns an lvalue or a const rvalue pair<key\_type, mapped\_type>, then both key\_type and mapped\_type shall be CopyConstructible.

4 *Effects:* Constructs an empty multimap using the specified comparison object and allocator, and inserts elements from the range [first, last).

5 *Complexity:* Linear in  $N$  if the range [first, last) is already sorted using comp and otherwise  $N \log N$ , where  $N$  is last - first.

### 23.3.2.2 multimap modifiers

[multimap.modifiers]

```

template <class P>
 requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
iterator insert(P&& x);
template <class P>
 requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
iterator insert(const_iterator position, P&& x);

```

1 *Requires:* P shall be convertible to value\_type.

If P is instantiated as a reference type, then the argument x is copied from. Otherwise x is considered to be an rvalue as it is converted to value\_type and inserted into the map. Specifically, in such cases CopyConstructible is not required of key\_type or mapped\_type unless the conversion from P specifically requires it (e.g. if P is a tuple<const key\_type, mapped\_type>, then key\_type must be CopyConstructible). The signature taking InputIterator parameters does not require CopyConstructible of either key\_type or mapped\_type if the dereferenced InputIterator returns a non-const rvalue pair<key\_type, mapped\_type>. Otherwise CopyConstructible is required for both key\_type and mapped\_type.

### 23.3.2.3 multimap operations

[multimap.ops]

```

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

```

```

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

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

```

- 1 The find, lower\_bound, upper\_bound, and equal\_range member functions each have two versions, one const and one non-const. In each case the behavior of the two versions is identical except that the const version returns a const\_iterator and the non-const version an iterator (23.1.2).

#### 23.3.2.4 multimap specialized algorithms

[multimap.special]

```

template <classValueType Key, classValueType T, class Compare, class Allocator>
 void swap(multimap<Key,T,Compare,Allocator>& x,
 multimap<Key,T,Compare,Allocator>& y);
template <classValueType Key, classValueType T, class Compare, class Allocator>
 void swap(multimap<Key,T,Compare,Allocator>&& x,
 multimap<Key,T,Compare,Allocator>&& y);
template <classValueType Key, classValueType T, class Compare, class Allocator>
 void swap(multimap<Key,T,Compare,Allocator>& x,
 multimap<Key,T,Compare,Allocator>&& y);

```

- 1 *Effects:*  
     x.swap(y);

#### 23.3.3 Class template set

[set]

- 1 A set is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves. Class set supports bidirectional iterators.
- 2 A set satisfies all of the requirements of a container, of a reversible container (23.1), of an associative container (23.1.2), and of an allocator-aware container (Table ??). A set also provides most operations described in (23.1.2) for unique keys. This means that a set supports the a\_uniq operations in (23.1.2) but not the a\_eq operations. For a set<Key> both the key\_type and value\_type are Key. Descriptions are provided here only for operations on set that are not described in one of these tables and for operations where there is additional semantic information.

```

namespace std {
 template <classValueType Key, classPredicate<auto, Key, Key> Compare = less<Key>,
 classAllocator Allocator = allocator<Key> >
 requires NothrowDestructible<Key> && CopyConstructible<Compare>
 && AllocatableElement<Alloc, Compare, const Compare&>
 && AllocatableElement<Alloc, Compare, Compare&&>
 class set {
 public:
 // types:

```

```

typedef Key key_type;
typedef Key value_type;
typedef Compare key_compare;
typedef Compare value_compare;
typedef Allocator allocator_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference const_reference;
typedef implementation-defined iterator; // See 23.1
typedef implementation-defined const_iterator; // See 23.1
typedef implementation-defined size_type; // See 23.1
typedef implementation-defined difference_type; // See 23.1
typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer const_pointer;
typedef reverse_iterator<iterator> reverse_iterator;
typedef reverse_iterator<const_iterator> const_reverse_iterator;

// 23.3.3.1 construct/copy/destroy:
explicit set(const Compare& comp = Compare(),
 const Allocator& = Allocator());
template <class InputIterator, InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 set(InputIterator first, InputIterator last,
 const Compare& comp = Compare(), const Allocator& = Allocator());
requires AllocatableElement<Alloc, value_type, const value_type&>
 set(const set<Key, Compare, Allocator>& x);
set(set<Key, Compare, Allocator>&& x);
set(const Allocator&);
requires AllocatableElement<Alloc, value_type, const value_type&>
 set(const set&, const Allocator&);
requires AllocatableElement<Alloc, value_type, value_type&&>
 set(set&&, const Allocator&);
requires AllocatableElement<Alloc, value_type, const value_type&>
 set(initializer_list<value_type>,
 const Compare& = Compare(),
 const Allocator& = Allocator());
~set();

requires AllocatableElement<Alloc, value_type, const value_type&>
 && CopyAssignable<value_type>
 set<Key, Compare, Allocator>& operator=(const set<Key, Compare, Allocator>& x);
set<Key, Compare, Allocator>& operator=(set<Key, Compare, Allocator>&& x);
requires AllocatableElement<Alloc, value_type, const value_type&>
 && CopyAssignable<value_type>
 set<Key, Compare, Allocator>& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// iterators:
iterator begin();
const_iterator begin() const;

```

```

iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
bool empty() const;
size_type size() const;
size_type max_size() const;

// modifiers:
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 iterator emplace_hint(const_iterator position, Args&&... args);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 pair<iterator, bool> insert(const value_type& x);
 requires AllocatableElement<Alloc, value_type, value_type&&>
 pair<iterator, bool> insert(value_type&& x);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 iterator insert(const_iterator position, const value_type& x);
 requires AllocatableElement<Alloc, value_type, value_type&&>
 iterator insert(const_iterator position, value_type&& x);
template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference> && MoveConstructible<value_type>
 void insert(InputIteratorIter first, InputIteratorIter last);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(set<Key, Compare, Allocator>&);
void clear();

// observers:
key_compare key_comp() const;
value_compare value_comp() const;

// set operations:

```

```

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

 size_type count(const key_type& x) const;

 iterator lower_bound(const key_type& x);
 const_iterator lower_bound(const key_type& x) const;

 iterator upper_bound(const key_type& x);
 const_iterator upper_bound(const key_type& x) const;

 pair<iterator,iterator> equal_range(const key_type& x);
 pair<const_iterator,const_iterator> equal_range(const key_type& x) const;
};

template <class EqualityComparable Key, class Compare, class Allocator>
 bool operator==(const set<Key,Compare,Allocator>& x,
 const set<Key,Compare,Allocator>& y);
template <class LessThanComparable Key, class Compare, class Allocator>
 bool operator< (const set<Key,Compare,Allocator>& x,
 const set<Key,Compare,Allocator>& y);
template <class EqualityComparable Key, class Compare, class Allocator>
 bool operator!=(const set<Key,Compare,Allocator>& x,
 const set<Key,Compare,Allocator>& y);
template <class LessThanComparable Key, class Compare, class Allocator>
 bool operator> (const set<Key,Compare,Allocator>& x,
 const set<Key,Compare,Allocator>& y);
template <class LessThanComparable Key, class Compare, class Allocator>
 bool operator>= (const set<Key,Compare,Allocator>& x,
 const set<Key,Compare,Allocator>& y);
template <class LessThanComparable Key, class Compare, class Allocator>
 bool operator<= (const set<Key,Compare,Allocator>& x,
 const set<Key,Compare,Allocator>& y);

// specialized algorithms:
template <class ValueType Key, class Compare, class Allocator>
 void swap(set<Key,Compare,Allocator>& x,
 set<Key,Compare,Allocator>& y);
template <class ValueType Key, class Compare, class Allocator>
 void swap(set<Key,Compare,Allocator>&& x,
 set<Key,Compare,Allocator>& y);
template <class ValueType Key, class Compare, class Allocator>
 void swap(set<Key,Compare,Allocator>& x,
 set<Key,Compare,Allocator>&& y);

template <class Key, class Compare, class Alloc>
 struct constructible_with_allocator_suffix<
 set<Key, Compare, Alloc>
 : true_type { };
}

```

## 23.3.3.1 set constructors, copy, and assignment

[set.cons]

```
explicit set(const Compare& comp = Compare(),
 const Allocator& = Allocator());
```

1 *Effects:* Constructs an empty set using the specified comparison objects and allocator.

2 *Complexity:* Constant.

```
template <class InputIterator, InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
set(InputIterator first, InputIterator last,
 const Compare& comp = Compare(), const Allocator& = Allocator());
```

3 *Effects:* Constructs an empty set using the specified comparison object and allocator, and inserts elements from the range  $[first, last)$ .

4 *Requires:* If the iterator's dereference operator returns an lvalue or a non-const rvalue, then Key shall be CopyConstructible.

5 *Complexity:* Linear in  $N$  if the range  $[first, last)$  is already sorted using *comp* and otherwise  $N \log N$ , where  $N$  is  $last - first$ .

## 23.3.3.2 set specialized algorithms

[set.special]

```
template <class ValueType, Key, class Compare, class Allocator>
 void swap(set<Key, Compare, Allocator>& x,
 set<Key, Compare, Allocator>& y);
template <class ValueType, Key, class Compare, class Allocator>
 void swap(set<Key, Compare, Allocator>&& x,
 set<Key, Compare, Allocator>& y);
template <class ValueType, Key, class Compare, class Allocator>
 void swap(set<Key, Compare, Allocator>& x,
 set<Key, Compare, Allocator>&& y);
```

1 *Effects:*

```
x.swap(y);
```

## 23.3.4 Class template multiset

[multiset]

1 A multiset is an associative container that supports equivalent keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves. Class multiset supports bidirectional iterators.

2 A multiset satisfies all of the requirements of a container, of a reversible container (23.1), of an associative container (23.1.2), and of an allocator-aware container (Table ??). multiset also provides most operations described in (23.1.2) for duplicate keys. This means that a multiset supports the `a_eq` operations in (23.1.2) but not the `a_uniq` operations. For a multiset<Key> both the `key_type` and `value_type` are Key. Descriptions are provided here only for operations on multiset that are not described in one of these tables and for operations where there is additional semantic information.

```

namespace std {
 template <class ValueType Key, class Predicate<auto, Key, Key> Compare = less<Key>,
 class Allocator Allocator = allocator<Key> >
 requires NothrowDestructible<Key> && CopyConstructible<Compare>
 && AllocatableElement<Alloc, Compare, const Compare&>
 && AllocatableElement<Alloc, Compare, Compare&&>
 class multiset {
 public:
 // types:
 typedef Key key_type;
 typedef Key value_type;
 typedef Compare key_compare;
 typedef Compare value_compare;
 typedef Allocator allocator_type;
 typedef typename Allocator::reference reference;
 typedef typename Allocator::const_reference const_reference;
 typedef implementation-defined iterator; // See 23.1
 typedef implementation-defined const_iterator; // See 23.1
 typedef implementation-defined size_type; // See 23.1
 typedef implementation-defined difference_type; // See 23.1
 typedef typename Allocator::pointer pointer;
 typedef typename Allocator::const_pointer const_pointer;
 typedef reverse_iterator<iterator> reverse_iterator;
 typedef reverse_iterator<const_iterator> const_reverse_iterator;

 // construct/copy/destroy:
 explicit multiset(const Compare& comp = Compare(),
 const Allocator& = Allocator());
 template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 multiset(InputIteratorIter first, InputIteratorIter last,
 const Compare& comp = Compare(),
 const Allocator& = Allocator());
 requires AllocatableElement<Alloc, value_type, const value_type&>
 multiset(const multiset<Key, Compare, Allocator>& x);
 multiset(multiset<Key, Compare, Allocator>&& x);
 multiset(const Allocator&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 multiset(const multiset&, const Allocator&);
 requires AllocatableElement<Alloc, value_type, value_type&&>
 multiset(multiset&&, const Allocator&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 multiset(initializer_list<value_type>,
 const Compare& = Compare(),
 const Allocator& = Allocator());
 ~multiset();
 requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
 multiset<Key, Compare, Allocator>& operator=(const multiset<Key, Compare, Allocator>& x);
 multiset<Key, Compare, Allocator>& operator=(multiset<Key, Compare, Allocator>&& x);
 };
}

```

```

requires AllocatableElement<Alloc, value_type, const value_type&>
 && CopyAssignable<value_type>
 multiset<Key, Compare, Allocator>& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// iterators:
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;

const_iterator cbegin() const;
const_iterator cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
bool empty() const;
size_type size() const;
size_type max_size() const;

// modifiers:
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 iterator emplace(Args&&... args);
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 iterator emplace_hint(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, value_type, const value_type&>
 iterator insert(const value_type& x);
requires AllocatableElement<Alloc, value_type, value_type&&>
 iterator insert(value_type&& x);
requires AllocatableElement<Alloc, value_type, const value_type&>
 iterator insert(const_iterator position, const value_type& x);
requires AllocatableElement<Alloc, value_type, value_type&&>
 iterator insert(const_iterator position, value_type&& x);
template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference> && MoveConstructible<value_type>
 void insert(InputIteratorIter first, InputIteratorIter last);
requires AllocatableElement<Alloc, value_type, const value_type&>
 void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);

```

```

void swap(multiset<Key,Compare,Allocator>&&);
void clear();

// observers:
key_compare key_comp() const;
value_compare value_comp() const;

// set operations:
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;

size_type count(const key_type& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;

pair<iterator,iterator> equal_range(const key_type& x);
pair<const_iterator,const_iterator> equal_range(const key_type& x) const;
};

template <class EqualityComparable Key, class Compare, class Allocator>
 bool operator==(const multiset<Key,Compare,Allocator>& x,
 const multiset<Key,Compare,Allocator>& y);
template <class LessThanComparable Key, class Compare, class Allocator>
 bool operator< (const multiset<Key,Compare,Allocator>& x,
 const multiset<Key,Compare,Allocator>& y);
template <class EqualityComparable Key, class Compare, class Allocator>
 bool operator!=(const multiset<Key,Compare,Allocator>& x,
 const multiset<Key,Compare,Allocator>& y);
template <class LessThanComparable Key, class Compare, class Allocator>
 bool operator> (const multiset<Key,Compare,Allocator>& x,
 const multiset<Key,Compare,Allocator>& y);
template <class LessThanComparable Key, class Compare, class Allocator>
 bool operator>=(const multiset<Key,Compare,Allocator>& x,
 const multiset<Key,Compare,Allocator>& y);
template <class LessThanComparable Key, class Compare, class Allocator>
 bool operator<=(const multiset<Key,Compare,Allocator>& x,
 const multiset<Key,Compare,Allocator>& y);

// specialized algorithms:
template <class ValueType Key, class Compare, class Allocator>
 void swap(multiset<Key,Compare,Allocator>& x,
 multiset<Key,Compare,Allocator>& y);
template <class ValueType Key, class Compare, class Allocator>
 void swap(multiset<Key,Compare,Allocator>&& x,
 multiset<Key,Compare,Allocator>& y);
template <class ValueType Key, class Compare, class Allocator>

```

```

void swap(multiset<Key, Compare, Allocator>& x,
 multiset<Key, Compare, Allocator>&& y);

template <class Key, class Compare, class Alloc>
struct constructible_with_allocator_suffix<
multiset<Key, Compare, Alloc>->
 :: true_type { };
}

```

## 23.3.4.1 multiset constructors

[multiset.cons]

```

explicit multiset(const Compare& comp = Compare(),
 const Allocator& = Allocator());

```

1 *Effects:* Constructs an empty set using the specified comparison object and allocator.

2 *Complexity:* Constant.

```

template <class InputIterator, InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
multiset(InputIterator first, InputIterator last,
 const Compare& comp = Compare(),
 const Allocator& = Allocator());

```

3 *Requires:* If the iterator's dereference operator returns an lvalue or a const rvalue, then Key shall be CopyConstructible.

4 *Effects:* Constructs an empty multiset using the specified comparison object and allocator, and inserts elements from the range  $[first, last)$ .

5 *Complexity:* Linear in  $N$  if the range  $[first, last)$  is already sorted using  $comp$  and otherwise  $N \log N$ , where  $N$  is  $last - first$ .

## 23.3.4.2 multiset specialized algorithms

[multiset.special]

```

template <class ValueType, Key, class Compare, class Allocator>
 void swap(multiset<Key, Compare, Allocator>& x,
 multiset<Key, Compare, Allocator>& y);
template <class ValueType, Key, class Compare, class Allocator>
 void swap(multiset<Key, Compare, Allocator>&& x,
 multiset<Key, Compare, Allocator>& y);
template <class ValueType, Key, class Compare, class Allocator>
 void swap(multiset<Key, Compare, Allocator>& x,
 multiset<Key, Compare, Allocator>&& y);

```

1 *Effects:*

```

 x.swap(y);

```

## 23.4 Unordered associative containers

[unord]

1 Headers `<unordered_map>` and `<unordered_set>`:Header `<unordered_map>` synopsis

```

namespace std {
 // 23.4.1, class template unordered_map:
 template <class ValueType Key,
 class ValueType T,
 class Callable<auto, const Key&> Hash = hash<Key>,
 class Predicate<auto, Key, Key> Pred = equal_to<Key>,
 class Allocator Alloc = allocator<pair<const Key, T> > >
 requires NothrowDestructible<Key> && NothrowDestructible<T>
 && SameType<Hash::result_type, size_t>
 && CopyConstructible<Hash> && CopyConstructible<Pred>
 && AllocatableElement<Alloc, Pred, const Pred&>
 && AllocatableElement<Alloc, Pred, Pred&&>
 && AllocatableElement<Alloc, Hash, const Hash&>
 && AllocatableElement<Alloc, Hash, Hash&&>
 class unordered_map;

 // 23.4.2, class template unordered_multimap:
 template <class ValueType Key,
 class ValueType T,
 class Callable<auto, const Key&> Hash = hash<Key>,
 class Predicate<auto, Key, Key> Pred = equal_to<Key>,
 class Allocator Alloc = allocator<pair<const Key, T> > >
 requires NothrowDestructible<Key> && NothrowDestructible<T>
 && SameType<Hash::result_type, size_t>
 && CopyConstructible<Hash> && CopyConstructible<Pred>
 && AllocatableElement<Alloc, Pred, const Pred&>
 && AllocatableElement<Alloc, Pred, Pred&&>
 && AllocatableElement<Alloc, Hash, const Hash&>
 && AllocatableElement<Alloc, Hash, Hash&&>
 class unordered_multimap;

 template <class ValueType Key, class ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
 unordered_map<Key, T, Hash, Pred, Alloc>& y);
 template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
 unordered_map<Key, T, Hash, Pred, Alloc>&& y);
 template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_map<Key, T, Hash, Pred, Alloc>&& x,
 unordered_map<Key, T, Hash, Pred, Alloc>& y);

 template <class ValueType Key, class ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
 unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
 template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>

```

```

 void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
 unordered_multimap<Key, T, Hash, Pred, Alloc>&& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>&& x,
 unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
} // namespace std

```

**Header `<unordered_set>` synopsis**

```

namespace std {
 // 23.4.3, class template unordered_set:
 template <class ValueType Value,
 class Callable<auto, const Value&> Hash = hash<Value>,
 class Predicate<auto, Value, Value> class Pred = equal_to<Value>,
 class Allocator Alloc = allocator<Value> >
 requires NothrowDestructible<Value>
 && SameType<Hash::result_type, size_t>
 && CopyConstructible<Hash> && CopyConstructible<Pred>
 && AllocatableElement<Alloc, Pred, const Pred&>
 && AllocatableElement<Alloc, Pred, Pred&&>
 && AllocatableElement<Alloc, Hash, const Hash&>
 && AllocatableElement<Alloc, Hash, Hash&&>
 class unordered_set;

 // 23.4.4, class template unordered_multiset:
 template <class ValueType Value,
 class Callable<auto, const Value&> Hash = hash<Value>,
 class Predicate<auto, Value, Value> class Pred = equal_to<Value>,
 class Allocator Alloc = allocator<Value> >
 requires NothrowDestructible<Value>
 && SameType<Hash::result_type, size_t>
 && CopyConstructible<Hash> && CopyConstructible<Pred>
 && AllocatableElement<Alloc, Pred, const Pred&>
 && AllocatableElement<Alloc, Pred, Pred&&>
 && AllocatableElement<Alloc, Hash, const Hash&>
 && AllocatableElement<Alloc, Hash, Hash&&>
 class unordered_multiset;

 template <class ValueType Value, class Hash, class Pred, class Alloc>
 void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
 unordered_set<Value, Hash, Pred, Alloc>&& y);
 template <ValueType Value, class Hash, class Pred, class Alloc>
 void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
 unordered_set<Value, Hash, Pred, Alloc>&& y);
 template <ValueType Value, class Hash, class Pred, class Alloc>
 void swap(unordered_set<Value, Hash, Pred, Alloc>&& x,
 unordered_set<Value, Hash, Pred, Alloc>& y);

 template <class ValueType Value, class Hash, class Pred, class Alloc>
 void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,

```

```

 unordered_multiset<Value, Hash, Pred, Alloc>& y);
template <ValueType Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
 unordered_multiset<Value, Hash, Pred, Alloc>&& y);
template <ValueType Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>&& x,
 unordered_multiset<Value, Hash, Pred, Alloc>& y);
} // namespace std

```

### 23.4.1 Class template unordered\_map

[unord.map]

- 1 An unordered\_map is an unordered associative container that supports unique keys (an unordered\_map contains at most one of each key value) and that associates values of another type mapped\_type with the keys.
- 2 An unordered\_map satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table ??). It provides the operations described in the preceding requirements table for unique keys; that is, an unordered\_map supports the a\_uniq operations in that table, not the a\_eq operations. For an unordered\_map<Key, T> the key type is Key, the mapped type is T, and the value type is pair<const Key, T>.
- 3 This section only describes operations on unordered\_map that are not described in one of the requirement tables, or for which there is additional semantic information.

```

namespace std {
 template <classValueType Key,
 classValueType T,
 classCallable<auto, const Key&> Hash = hash<Key>,
 classPredicate<auto, Key, Key> Pred = equal_to<Key>,
 classAllocator Alloc = allocator<pair<const Key, T> > >
 requires NothrowDestructible<Key> && NothrowDestructible<T>
 && SameType<Hash::result_type, size_t>
 && CopyConstructible<Hash> && CopyConstructible<Pred>
 && AllocatableElement<Alloc, Pred, const Pred&&>
 && AllocatableElement<Alloc, Pred, Pred&&>
 && AllocatableElement<Alloc, Hash, const Hash&&>
 && AllocatableElement<Alloc, Hash, Hash&&>
 class unordered_map
 {
 public:
 // types
 typedef Key key_type;
 typedef pair<const Key, T> value_type;
 typedef T mapped_type;
 typedef Hash hasher;
 typedef Pred key_equal;
 typedef Alloc allocator_type;
 typedef typename allocator_type::pointer pointer;
 typedef typename allocator_type::const_pointer const_pointer;
 typedef typename allocator_type::reference reference;
 typedef typename allocator_type::const_reference const_reference;
 typedef implementation-defined size_type;

```

```

typedef implementation-defined difference_type;

typedef implementation-defined iterator;
typedef implementation-defined const_iterator;
typedef implementation-defined local_iterator;
typedef implementation-defined const_local_iterator;

// construct/destroy/copy
explicit unordered_map(size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());

template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 unordered_map(InputIteratorIter f, InputIteratorIter l,
 size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
 requires AllocatableElement<Alloc, value_type, const value_type&>
 unordered_map(const unordered_map&);
 requires AllocatableElement<Alloc, value_type, value_type&&>
 unordered_map(unordered_map&&);
 unordered_map(const Allocator&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 unordered_map(const unordered_map&, const Allocator&);
 requires AllocatableElement<Alloc, value_type, value_type&&>
 unordered_map(unordered_map&&, const Allocator&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 unordered_map(initializer_list<value_type>,
 size_type = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
~unordered_map();
 requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
 unordered_map& operator=(const unordered_map&);
 requires AllocatableElement<Alloc, value_type, value_type&&> && MoveAssignable<value_type>
 unordered_map& operator=(unordered_map&&);
 requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
 unordered_map& operator=(initializer_list<value_type>);
 allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators

```

```

iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 iterator emplace_hint(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, value_type, const value_type&>
 pair<iterator, bool> insert(const value_type& obj);
template <class P>
 requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
 pair<iterator, bool> insert(P&& obj);
requires AllocatableElement<Alloc, value_type, const value_type&>
 iterator insert(const_iterator hint, const value_type& obj);
template <class P>
 requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
 pair<iterator, bool> insert(const_iterator hint, P&& obj);
template <class InputIterator, Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 void insert(InputIterator first, InputIterator last);
requires AllocatableElement<Alloc, value_type, const value_type&>
 void insert(initializer_list<value_type>);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_map&&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

requires AllocatableElement<Alloc, value_type, const key_type&, mapped_type&&>
 && AllocatableElement<Alloc, mapped_type>

```

```

 mapped_type& operator[](const key_type& k);
 requires AllocatableElement<Alloc, value_type, key_type&&, mapped_type&&>
 && AllocatableElement<Alloc, mapped_type>
 mapped_type& operator[](key_type&& k);
 mapped_type& at(const key_type& k);
 const mapped_type& at(const key_type& k) const;

 // bucket interface
 size_type bucket_count() const;
 size_type max_bucket_count() const;
 size_type bucket_size(size_type n);
 size_type bucket(const key_type& k) const;
 local_iterator begin(size_type n) const;
 const_local_iterator begin(size_type n) const;
 local_iterator end(size_type n);
 const_local_iterator end(size_type n) const;

 // hash policy
 float load_factor() const;
 float max_load_factor() const;
 void max_load_factor(float z);
 requires MoveConstructible<value_type> void rehash(size_type n);
};

template <class ValueType Key, class ValueType T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
 unordered_map<Key, T, Hash, Pred, Alloc>& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
 unordered_map<Key, T, Hash, Pred, Alloc>&& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>&& x,
 unordered_map<Key, T, Hash, Pred, Alloc>& y);

template <class Key, class T, class Hash, class Pred, class Alloc>
struct constructible_with_allocator_suffix<
unordered_map<Key, T, Hash, Pred, Compare, Alloc> ->
:- true_type {-};
}

```

#### 23.4.1.1 unordered\_map constructors

[unord.map.cnstr]

```

explicit unordered_map(size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());

```

- 1 *Effects:* Constructs an empty unordered\_map using the specified hash function, key equality function, and allocator, and using at least  $n$  buckets. If  $n$  is not provided, the number of buckets is implementation defined. max\_load\_factor() returns 1.0.

2 *Complexity:* Constant.

```
template <class InputIterator, InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
unordered_map(InputIterator f, InputIterator l,
 size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty unordered\_map using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range  $[f, l)$ . `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

#### 23.4.1.2 unordered\_map element access

[unord.map.elem]

```
requires AllocatableElement<Alloc, value_type, key_type&&, mapped_type&&>
 && AllocatableElement<Alloc, mapped_type>
mapped_type& operator[](const key_type& k);
```

```
requires AllocatableElement<Alloc, value_type, key_type&&, mapped_type&&>
 && AllocatableElement<Alloc, mapped_type>
mapped_type& operator[](key_type&& k);
```

1 *Effects:* If the unordered\_map does not already contain an element whose key is equivalent to *k*, inserts the value pair<const key\_type, mapped\_type>(k, mapped\_type()) or pair<const key\_type, mapped\_type>(move(k), mapped\_type()), respectively.

2 *Returns:* A reference to `x.second`, where *x* is the (unique) element whose key is equivalent to *k*.

```
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;
```

3 *Returns:* A reference to `x.second`, where *x* is the (unique) element whose key is equivalent to *k*.

4 *Throws:* An exception object of type `out_of_range` if no such element is present.

#### 23.4.1.3 unordered\_map swap

[unord.map.swap]

```
template <class ValueType Key, class ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
 unordered_map<Key, T, Hash, Pred, Alloc>& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
 unordered_map<Key, T, Hash, Pred, Alloc>&& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_map<Key, T, Hash, Pred, Alloc>&& x,
```

```
unordered_map<Key, T, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y)`.

### 23.4.2 Class template `unordered_multimap`

[unord.multimap]

- 1 An `unordered_multimap` is an unordered associative container that supports equivalent keys (an `unordered_multimap` may contain multiple copies of each key value) and that associates values of another type `mapped_type` with the keys.
- 2 An `unordered_multimap` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table ??). It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multimap` supports the `a_eq` operations in that table, not the `a_uniq` operations. For an `unordered_multimap<Key, T>` the key type is `Key`, the mapped type is `T`, and the value type is `pair<const Key, T>`.
- 3 This section only describes operations on `unordered_multimap` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
 template <classValueType Key,
 classValueType T,
 classCallable<auto, const Key&> Hash = hash<Key>,
 classPredicate<auto, Key, Key> Pred = equal_to<Key>,
 classAllocator Alloc = allocator<pair<const Key, T> > >
 requires NothrowDestructible<Key> && NothrowDestructible<T>
 && SameType<Hash::result_type, size_t>
 && CopyConstructible<Hash> && CopyConstructible<Pred>
 && AllocatableElement<Alloc, Pred, const Pred&&>
 && AllocatableElement<Alloc, Pred, Pred&&>
 && AllocatableElement<Alloc, Hash, const Hash&&>
 && AllocatableElement<Alloc, Hash, Hash&&>
 class unordered_multimap
 {
 public:
 // types
 typedef Key key_type;
 typedef pair<const Key, T> value_type;
 typedef T mapped_type;
 typedef Hash hasher;
 typedef Pred key_equal;
 typedef Alloc allocator_type;
 typedef typename allocator_type::pointer pointer;
 typedef typename allocator_type::const_pointer const_pointer;
 typedef typename allocator_type::reference reference;
 typedef typename allocator_type::const_reference const_reference;
 typedef implementation-defined size_type;
 typedef implementation-defined difference_type;

 typedef implementation-defined iterator;
 typedef implementation-defined const_iterator;
 };
}
```

```

typedef implementation-defined local_iterator;
typedef implementation-defined const_local_iterator;

// construct/destroy/copy
explicit unordered_multimap(size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());

template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
unordered_multimap(InputIteratorIter f, InputIteratorIter l,
 size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
requires AllocatableElement<Alloc, value_type, const value_type>
unordered_multimap(const unordered_multimap&);
requires AllocatableElement<Alloc, value_type, value_type&&>
unordered_multimap(unordered_multimap&&);
unordered_multimap(const Allocator&);
requires AllocatableElement<Alloc, value_type, const value_type>
unordered_multimap(const unordered_multimap&, const Allocator&);
requires AllocatableElement<Alloc, value_type, value_type&&>
unordered_multimap(unordered_multimap&&, const Allocator&);
requires AllocatableElement<Alloc, value_type, const value_type>
unordered_multimap(initializer_list<value_type>,
 size_type = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
~unordered_multimap();
requires AllocatableElement<Alloc, value_type, const value_type> && CopyAssignable<value_type>
unordered_multimap& operator=(const unordered_multimap&);
requires AllocatableElement<Alloc, value_type, value_type&&> && MoveAssignable<value_type>
unordered_multimap& operator=(unordered_multimap&&);
requires AllocatableElement<Alloc, value_type, const value_type> && CopyAssignable<value_type>
unordered_multimap& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

```

```

const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 iterator emplace(Args&&... args);
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 iterator emplace_hint(const_iterator position, Args&&... args);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 iterator insert(const value_type& obj);
template <class P>
 requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
 iterator insert(P&& obj);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 iterator insert(const_iterator hint, const value_type& obj);
template <class P>
 requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
 iterator insert(const_iterator hint, P&& obj);
template <class InputIterator> InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 void insert(InputIterator first, InputIterator last);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_multimap&&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n);
size_type bucket(const key_type& k) const;

```

```

 local_iterator begin(size_type n) const;
 const_local_iterator begin(size_type n) const;
 local_iterator end(size_type n);
 const_local_iterator end(size_type n) const;

 // hash policy
 float load_factor() const;
 float max_load_factor() const;
 void max_load_factor(float z);
 requires MoveConstructible<value_type> void rehash(size_type n);
};

template <class ValueType Key, class ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
 unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
 unordered_multimap<Key, T, Hash, Pred, Alloc>&& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>&& x,
 unordered_multimap<Key, T, Hash, Pred, Alloc>& y);

template <class Key, class T, class Hash, class Pred, class Alloc>
 struct constructible_with_allocator_suffix<
 unordered_multimap<Key, T, Hash, Pred, Alloc> ->
 ::true_type { };
}

```

#### 23.4.2.1 unordered\_multimap constructors

[unord.multimap.cnstr]

```

explicit unordered_multimap(size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());

```

1 *Effects:* Constructs an empty unordered\_multimap using the specified hash function, key equality function, and allocator, and using at least  $n$  buckets. If  $n$  is not provided, the number of buckets is implementation defined. max\_load\_factor() returns 1.0.

2 *Complexity:* Constant.

```

template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 unordered_multimap(InputIteratorIter f, InputIteratorIter l,
 size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());

```

3 *Effects:* Constructs an empty unordered\_multimap using the specified hash function, key equality function, and

allocator, and using at least  $n$  buckets. (If  $n$  is not provided, the number of buckets is implementation defined.) Then inserts elements from the range  $[f, l)$ . `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

### 23.4.2.2 unordered\_multimap swap

[unord.multimap.swap]

```
template <class ValueType Key, class ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
 unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
 unordered_multimap<Key, T, Hash, Pred, Alloc>&& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
 void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>&& x,
 unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y)`.

### 23.4.3 Class template unordered\_set

[unord.set]

- 1 An `unordered_set` is an unordered associative container that supports unique keys (an `unordered_set` contains at most one of each key value) and in which the elements' keys are the elements themselves.
- 2 An `unordered_set` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table ??). It provides the operations described in the preceding requirements table for unique keys; that is, an `unordered_set` supports the `a_uniq` operations in that table, not the `a_eq` operations. For an `unordered_set<Value>` the key type and the value type are both `Value`. The iterator and `const_iterator` types are both `const_iterator` types. It is unspecified whether they are the same type.
- 3 This section only describes operations on `unordered_set` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
 template <class ValueType Value,
 class Callable<auto, const Value&> Hash = hash<Value>,
 class Predicate<auto, Value, Value> class Pred = equal_to<Value>,
 class Allocator Alloc = allocator<Value> >
 requires NothrowDestructible<Value>
 && SameType<Hash::result_type, size_t>
 && CopyConstructible<Hash> && CopyConstructible<Pred>
 && AllocatableElement<Alloc, Pred, const Pred&>
 && AllocatableElement<Alloc, Pred, Pred&&>
 && AllocatableElement<Alloc, Hash, const Hash&>
 && AllocatableElement<Alloc, Hash, Hash&&>
 class unordered_set
 {
 public:
 // types
```

```

typedef Value key_type;
typedef Value value_type;
typedef Hash hasher;
typedef Pred key_equal;
typedef Alloc allocator_type;
typedef typename allocator_type::pointer pointer;
typedef typename allocator_type::const_pointer const_pointer;
typedef typename allocator_type::reference reference;
typedef typename allocator_type::const_reference const_reference;
typedef implementation-defined size_type;
typedef implementation-defined difference_type;

typedef implementation-defined iterator;
typedef implementation-defined const_iterator;
typedef implementation-defined local_iterator;
typedef implementation-defined const_local_iterator;

// construct/destroy/copy
explicit unordered_set(size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());

template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 unordered_set(InputIteratorIter f, InputIteratorIter l,
 size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
 requires AllocatableElement<Alloc, value_type, const value_type&>
 unordered_set(const unordered_set&);
 requires AllocatableElement<Alloc, value_type, value_type&&>
 unordered_set(unordered_set&&);
 unordered_set(const Allocator&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 unordered_set(const unordered_set&, const Allocator&);
 requires AllocatableElement<Alloc, value_type, value_type&&>
 unordered_set(unordered_set&&, const Allocator&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 unordered_set(initializer_list<value_type>,
 size_type = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
~unordered_set();
 requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
 unordered_set& operator=(const unordered_set&);
 requires AllocatableElement<Alloc, value_type, value_type&&> && MoveAssignable<value_type>
 unordered_set& operator=(unordered_set&&);

```

```

requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
unordered_set& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
 requires AllocatableElement<Alloc, value_type, Args&&...>
 iterator emplace_hint(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, value_type, const value_type&>
 pair<iterator, bool> insert(const value_type& obj);
requires AllocatableElement<Alloc, value_type, value_type&&>
 pair<iterator, bool> insert(value_type&& obj);
requires AllocatableElement<Alloc, value_type, const value_type&>
 iterator insert(const_iterator hint, const value_type& obj);
requires AllocatableElement<Alloc, value_type, value_type&&>
 iterator insert(const_iterator hint, value_type&& obj);
template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 void insert(InputIteratorIter first, InputIteratorIter last);
requires AllocatableElement<Alloc, value_type, const value_type&>
 void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_set&&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

```

```

// lookup
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
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;

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
requires MoveConstructible<value_type> void rehash(size_type n);
};

template <class Value, class Hash, class Pred, class Alloc>
void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
 unordered_set<Value, Hash, Pred, Alloc>& y);
template <Value Type Value, class Hash, class Pred, class Alloc>
void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
 unordered_set<Value, Hash, Pred, Alloc>&& y);
template <Value Type Value, class Hash, class Pred, class Alloc>
void swap(unordered_set<Value, Hash, Pred, Alloc>&& x,
 unordered_set<Value, Hash, Pred, Alloc>& y);

template <class Value, class Hash, class Pred, class Alloc>
struct constructible_with_allocator_suffix<
map<Value, Hash, Pred, Alloc>->
:- true_type{-};
}

```

### 23.4.3.1 unordered\_set constructors

[unord.set.cnstr]

```

explicit unordered_set(size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());

```

- 1 *Effects:* Constructs an empty unordered\_set using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. max\_load\_factor() returns 1.0.

2 *Complexity:* Constant.

```
template <class InputIterator, InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
unordered_set(InputIterator f, InputIterator l,
 size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty `unordered_set` using the specified hash function, key equality function, and allocator, and using at least  $n$  buckets. (If  $n$  is not provided, the number of buckets is implementation defined.) Then inserts elements from the range  $[f, l)$ . `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

### 23.4.3.2 unordered\_set swap

[unord.set.swap]

```
template <class ValueType, class Hash, class Pred, class Alloc>
 void swap(unordered_set<ValueType, Hash, Pred, Alloc>& x,
 unordered_set<ValueType, Hash, Pred, Alloc>& y);
template <ValueType Value, class Hash, class Pred, class Alloc>
 void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
 unordered_set<Value, Hash, Pred, Alloc>&& y);
template <ValueType Value, class Hash, class Pred, class Alloc>
 void swap(unordered_set<Value, Hash, Pred, Alloc>&& x,
 unordered_set<Value, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y)`.

### 23.4.4 Class template unordered\_multiset

[unord.multiset]

- 1 An `unordered_multiset` is an unordered associative container that supports equivalent keys (an `unordered_multiset` may contain multiple copies of the same key value) and in which each element's key is the element itself.
- 2 An `unordered_multiset` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table ??). It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multiset` supports the `a_eq` operations in that table, not the `a_uniq` operations. For an `unordered_multiset<Value>` the key type and the value type are both `Value`. The iterator and `const_iterator` types are both `const_iterator` types. It is unspecified whether they are the same type.
- 3 This section only describes operations on `unordered_multiset` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
 template <class ValueType, Value,
 class Callable<auto, const Value&> Hash = hash<Value>,
 class Predicate<auto, Value, Value> Pred = equal_to<Value>,
 class Allocator Alloc = allocator<Value> >
```

```

requires NothrowDestructible<Value>
 && SameType<Hash::result_type, size_t>
 && CopyConstructible<Hash> && CopyConstructible<Pred>
 && AllocatableElement<Alloc, Pred, const Pred&>
 && AllocatableElement<Alloc, Pred, Pred&&>
 && AllocatableElement<Alloc, Hash, const Hash&>
 && AllocatableElement<Alloc, Hash, Hash&&>
class unordered_multiset
{
public:
 // types
 typedef Value key_type;
 typedef Value value_type;
 typedef Hash hasher;
 typedef Pred key_equal;
 typedef Alloc allocator_type;
 typedef typename allocator_type::pointer pointer;
 typedef typename allocator_type::const_pointer const_pointer;
 typedef typename allocator_type::reference reference;
 typedef typename allocator_type::const_reference const_reference;
 typedef implementation-defined size_type;
 typedef implementation-defined difference_type;

 typedef implementation-defined iterator;
 typedef implementation-defined const_iterator;
 typedef implementation-defined local_iterator;
 typedef implementation-defined const_local_iterator;

 // construct/destroy/copy
 explicit unordered_multiset(size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());

 template <class InputIterator InputIterator Iter>
 requires AllocatableElement<Alloc, value_type, Iter::reference>
 && MoveConstructible<value_type>
 unordered_multiset(InputIteratorIter f, InputIteratorIter l,
 size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
 requires AllocatableElement<Alloc, value_type, const value_type&>
 unordered_multiset(const unordered_multiset&);
 requires AllocatableElement<Alloc, value_type, value_type&&>
 unordered_multiset(unordered_multiset&&);
 unordered_multiset(const Allocator&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
 unordered_multiset(const unordered_multiset&, const Allocator&);
 requires AllocatableElement<Alloc, value_type, value_type&&>
 unordered_multiset(unordered_multiset&&, const Allocator&);

```

```

requires AllocatableElement<Alloc, value_type, const value_type&>
unordered_multiset(initializer_list<value_type>,
 size_type = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
~unordered_multiset();
requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
unordered_multiset& operator=(const unordered_multiset&);
requires AllocatableElement<Alloc, value_type, value_type&&> && MoveAssignable<value_type>
unordered_multiset& operator=(unordered_multiset&&);
requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
unordered_multiset& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args>
requires AllocatableElement<Alloc, value_type, Args&&...>
iterator emplace(Args&&... args);
template <class... Args>
requires AllocatableElement<Alloc, value_type, Args&&...>
iterator emplace_hint(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, value_type, const value_type&>
iterator insert(const value_type& obj);
requires AllocatableElement<Alloc, value_type, value_type&&>
iterator insert(value_type&& obj);
requires AllocatableElement<Alloc, value_type, const value_type&>
iterator insert(const_iterator hint, const value_type& obj);
requires AllocatableElement<Alloc, value_type, value_type&&>
iterator insert(const_iterator hint, value_type&& obj);
template <class InputIterator InputIterator Iter>
requires AllocatableElement<Alloc, value_type, Iter::value_type>
&& MoveConstructible<value_type>
void insert(InputIteratorIter first, InputIteratorIter last);
requires AllocatableElement<Alloc, value_type, const value_type&>
void insert(initializer_list<value_type>);

```

```

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_multiset&&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n);
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n) const;
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
requires MoveConstructible<value_type> void rehash(size_type n);
};

template <class Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
 unordered_multiset<Value, Hash, Pred, Alloc>& y);
template <Value Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
 unordered_multiset<Value, Hash, Pred, Alloc>&& y);
template <Value Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>&& x,
 unordered_multiset<Value, Hash, Pred, Alloc>& y);

template <class Value, class Hash, class Pred, class Alloc>
struct constructible_with_allocator_suffix<
 unordered_multiset<Value, Hash, Pred, Alloc>->
 :: true_type { };
}

```

## 23.4.4.1 unordered\_multiset constructors

[unord.multiset.cnstr]

```
explicit unordered_multiset(size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty unordered\_multiset using the specified hash function, key equality function, and allocator, and using at least  $n$  buckets. If  $n$  is not provided, the number of buckets is implementation defined. max\_load\_factor() returns 1.0.

2 *Complexity:* Constant.

```
template <class InputIterator, InputIterator Iter>
requires AllocatableElement<Alloc, value_type, Iter::reference>
&& MoveConstructible<value_type>
unordered_multiset(InputIterator f, InputIterator l,
 size_type n = implementation-defined,
 const hasher& hf = hasher(),
 const key_equal& eql = key_equal(),
 const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty unordered\_multiset using the specified hash function, key equality function, and allocator, and using at least  $n$  buckets. (If  $n$  is not provided, the number of buckets is implementation defined.) Then inserts elements from the range  $[f, l)$ . max\_load\_factor() returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

## 23.4.4.2 unordered\_multiset swap

[unord.multiset.swap]

```
template <class ValueType Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
 unordered_multiset<Value, Hash, Pred, Alloc>& y);
template <ValueType Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
 unordered_multiset<Value, Hash, Pred, Alloc>&& y);
template <ValueType Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>&& x,
 unordered_multiset<Value, Hash, Pred, Alloc>& y);
```

1 *Effects:* x.swap(y);

## Bibliography

- [1] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek. Proposed wording for concepts (revision 8). Technical Report N2741=08-0251, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2008.