Document Number: P1463R0
Date: 2019-01-20
Reply to: Marshall Clow
CppAlliance
mclow.lists@gmail.com

Mandating the Standard Library: Clause 21 - Containers library

With the adoption of P0788R3, we have a new way of specifying requirements for the library clauses of the standard. This is one of a series of papers reformulating the requirements into the new format. This effort was strongly influenced by the informational paper P1369R0.

The changes in this series of papers fall into four broad categories.

— Change "participate in overload resolution" wording into "Constraints" elements

— Change "Requires" elements into either "Mandates" or "Expects", depending (mostly) on whether or not they can be checked at compile time.

— Drive-by fixes (hopefully very few)

This paper covers Clause 21 (Containers)

As a drive-by fix, (Thanks Tim!) a few places that said "`...::propagate_on_container_move_assignment` is true" now say "`...::propagate_on_container_move_assignment::value` is true"

The entire clause is reproduced here, but the changes are confined to a few sections:

— container.requirements.general 21.2.1
— sequence.reqmts 21.2.3
— container.node.cons 21.2.4.2
— container.node.observers 21.2.4.4
— container.node.modifiers 21.2.4.5
— associative.reqmts 21.2.6
— unord.req 21.2.7
— array.cons 21.3.7.2
— array.special 21.3.7.4
— array.tuple 21.3.7.6
— deque.cons 21.3.8.2
— deque.capacity 21.3.8.3
— forwardlist.cons 21.3.9.2
— forwardlist.modifiers 21.3.9.5
— forwardlist.ops 21.3.9.6
— list.cons 21.3.10.2
— list.capacity 21.3.10.3
— list.ops 21.3.10.5
— vector.cons 21.3.11.2
— vector.capacity 21.3.11.3
— map.modifiers 21.4.4.4
— multimap.modifiers 21.4.5.3
— unord.map.modifiers 21.5.4.4
— unord.multimap.modifiers 21.5.5.3
— queue.special 21.6.4.5
— priqueue.cons 21.6.5.2
— priqueue.special 21.6.5.5
— stack.special 21.6.6.5
— span.cons 21.7.3.2
— span.sub 21.7.3.3
— span.elem 21.7.3.5
— span.objectrep 21.7.3.7

Help for the editors: The changes here can be viewed as latex sources with the following commands

```
git clone git@github.com:mclow/mandate.git
cd mandate
git diff master..chapter21 containers.tex
```

21 Containers library [containers]

21.1 General [containers.general]

1 This Clause describes components that C++ programs may use to organize collections of information.

2 The following subclauses describe container requirements, and components for sequence containers and associative containers, as summarized in Table 61.

Table 61 — Containers library summary

	Subclause	Header(s)
21.2	Requirements	

21.3	Sequence containers	`<array>`
		`<deque>`
		`<forward_list>`
		`<list>`
		`<vector>`

21.4	Associative containers	`<map>` `<set>`
21.5	Unordered associative containers	`<unordered_map>` `<unordered_set>`
21.6	Container adaptors	`<queue>` `<stack>`
21.7	Views	`<span>`

21.2 Container requirements [container.requirements]

21.2.1 General container requirements [container.requirements.general]

1 Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.

2 All of the complexity requirements in this Clause are stated solely in terms of the number of operations on the contained objects. [*Example*: The copy constructor of type `vector<vector<int>>` has linear complexity, even though the complexity of copying each contained `vector<int>` is itself linear. — *end example*]

3 For the components affected by this subclause that declare an `allocator_type`, objects stored in these components shall be constructed using the function `allocator_traits<allocator_type>::rebind_traits<U>::construct` and destroyed using the function `allocator_traits<allocator_type>::rebind_traits<U>::destroy` (??), where `U` is either `allocator_type::value_type` or an internal type used by the container. These functions are called only for the container's element type, not for internal types used by the container. [*Note*: This means, for example, that a node-based container might need to construct nodes containing aligned buffers and call `construct` to place the element into the buffer. — *end note*]

4 In Tables 62, 63, and 64 `X` denotes a container class containing objects of type `T`, `a` and `b` denote values of type `X`, `u` denotes an identifier, `r` denotes a non-const value of type `X`, and `rv` denotes a non-const rvalue of type `X`.

Table 62 — Container requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition	Complexity
`X::value_type`	`T`		~~*Requires:*~~ *Expects:* `T` is *Cpp17Erasable* from `X` (see 21.2.1, below)	compile time
`X::reference`	`T&`			compile time

Table 62 — Container requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition	Complexity
`X::const_reference`	`const T&`			compile time
`X::iterator`	iterator type whose value type is `T`		any iterator category that meets the forward iterator requirements. convertible to `X::const_iterator`.	compile time
`X::const_iterator`	constant iterator type whose value type is `T`		any iterator category that meets the forward iterator requirements.	compile time
`X::difference_type`	signed integer type		is identical to the difference type of `X::iterator` and `X::const_iterator`	compile time
`X::size_type`	unsigned integer type		`size_type` can represent any non-negative value of `difference_type`	compile time
`X u;`			*Ensures:* `u.empty()`	constant
`X()`			*Ensures:* `X().empty()`	constant
`X(a)`			~~*Requires:*~~ *Expects:* `T` is *Cpp17CopyInsertable* into `X` (see below). *Ensures:* `a == X(a)`.	linear
`X u(a);` `X u = a;`			~~*Requires:*~~ *Expects:* `T` is *Cpp17CopyInsertable* into `X` (see below). *Ensures:* `u == a`	linear
`X u(rv);` `X u = rv;`			*Ensures:* `u` ~~shall be~~is equal to the value that `rv` had before this construction	(Note B)
`a = rv`	`X&`	All existing elements of `a` are either move assigned to or destroyed	*Ensures:* `a` ~~shall be~~is equal to the value that `rv` had before this assignment	linear
`a.~X()`	`void`		the destructor is applied to every element of `a`; any memory obtained is deallocated.	linear
`a.begin()`	`iterator`; `const_iterator` for constant `a`			constant

Table 62 — Container requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition	Complexity
`a.end()`	`iterator`; `const_iterator` for constant `a`			constant
`a.cbegin()`	`const_iterator`	`const_cast<X const&>(a).begin();`		constant
`a.cend()`	`const_iterator`	`const_cast<X const&>(a).end();`		constant
`a == b`	convertible to `bool`	`==` is an equivalence relation. `equal(a.begin(), a.end(), b.begin(), b.end())`	~~*Requires:*~~ *Expects:* `T` ~~is~~meets the *Cpp17-EqualityComparable* requirements	Constant if `a.size() != b.size()`, linear otherwise
`a != b`	convertible to `bool`	Equivalent to `!(a == b)`		linear
`a.swap(b)`	`void`		exchanges the contents of `a` and `b`	(Note A)
`swap(a, b)`	`void`	Equivalent to `a.swap(b)`		(Note A)
`r = a`	`X&`		*Ensures:* `r == a`.	linear
`a.size()`	`size_type`	`distance(a.begin(), a.end())`		constant
`a.max_size()`	`size_type`	`distance(begin(), end())` for the largest possible container		constant
`a.empty()`	convertible to `bool`	`a.begin() == a.end()`		constant

Those entries marked "(Note A)" or "(Note B)" have linear complexity for `array` and have constant complexity for all other standard containers. [*Note*: The algorithm `equal()` is defined in **??**. — *end note*]

5 The member function `size()` returns the number of elements in the container. The number of elements is defined by the rules of constructors, inserts, and erases.

6 `begin()` returns an iterator referring to the first element in the container. `end()` returns an iterator which is the past-the-end value for the container. If the container is empty, then `begin() == end()`.

7 In the expressions

```
i == j
i != j
i < j
i <= j
i >= j
i > j
i - j
```

where `i` and `j` denote objects of a container's `iterator` type, either or both may be replaced by an object of the container's `const_iterator` type referring to the same element with no change in semantics.

8 Unless otherwise specified, all containers defined in this clause obtain memory using an allocator (see **??**). [*Note*: In particular, containers and iterators do not store references to allocated elements other than through the allocator's pointer type, i.e., as objects of type `P` or `pointer_traits<P>::template rebind<`*`unspecified`*`>`, where `P` is `allocator_traits<allocator_type>::pointer`. — *end note*] Copy constructors for these container types obtain an allocator by calling `allocator_traits<allocator_type>::select_on_container_`

`copy_construction` on the allocator belonging to the container being copied. Move constructors obtain an allocator by move construction from the allocator belonging to the container being moved. Such move construction of the allocator shall not exit via an exception. All other constructors for these container types take a `const allocator_type&` argument. [*Note*: If an invocation of a constructor uses the default value of an optional allocator argument, then the allocator type must support value-initialization. — *end note*] A copy of this allocator is used for any memory allocation and element construction performed, by these constructors and by all member functions, during the lifetime of each container object or until the allocator is replaced. The allocator may be replaced only via assignment or `swap()`. Allocator replacement is performed by copy assignment, move assignment, or swapping of the allocator only if `allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value`, `allocator_traits<allocator_type>::propagate_on_container_move_assignment::value`, or `allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true` within the implementation of the corresponding container operation. In all container types defined in this Clause, the member `get_allocator()` returns a copy of the allocator used to construct the container or, if that allocator has been replaced, a copy of the most recent replacement.

9 The expression `a.swap(b)`, for containers `a` and `b` of a standard container type other than `array`, shall exchange the values of `a` and `b` without invoking any move, copy, or swap operations on the individual container elements. Lvalues of any `Compare`, `Pred`, or `Hash` types belonging to `a` and `b` shall be swappable and shall be exchanged by calling `swap` as described in **??**. If `allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`, then lvalues of type `allocator_type` shall be swappable and the allocators of `a` and `b` shall also be exchanged by calling `swap` as described in **??**. Otherwise, the allocators shall not be swapped, and the behavior is undefined unless `a.get_allocator() == b.get_allocator()`. Every iterator referring to an element in one container before the swap shall refer to the same element in the other container after the swap. It is unspecified whether an iterator with value `a.end()` before the swap will have value `b.end()` after the swap.

10 If the iterator type of a container belongs to the bidirectional or random access iterator categories (**??**), the container is called *reversible* and satisfies the additional requirements in Table 63.

Table 63 — Reversible container requirements

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::reverse_iterator`	iterator type whose value type is `T`	`reverse_iterator<iterator>`	compile time
`X::const_reverse_iterator`	constant iterator type whose value type is `T`	`reverse_iterator<const_iterator>`	compile time
`a.rbegin()`	`reverse_iterator`; `const_reverse_iterator` for constant `a`	`reverse_iterator(end())`	constant
`a.rend()`	`reverse_iterator`; `const_reverse_iterator` for constant `a`	`reverse_iterator(begin())`	constant
`a.crbegin()`	`const_reverse_iterator`	`const_cast<X const&>(a).rbegin()`	constant
`a.crend()`	`const_reverse_iterator`	`const_cast<X const&>(a).rend()`	constant

11 Unless otherwise specified (see 21.2.6.1, 21.2.7.1, 21.3.8.4, and 21.3.11.5) all container types defined in this Clause meet the following additional requirements:

(11.1) — if an exception is thrown by an `insert()` or `emplace()` function while inserting a single element, that function has no effects.

(11.2) — if an exception is thrown by a `push_back()`, `push_front()`, `emplace_back()`, or `emplace_front()` function, that function has no effects.

(11.3) — no `erase()`, `clear()`, `pop_back()` or `pop_front()` function throws an exception.

(11.4) — no copy constructor or assignment operator of a returned iterator throws an exception.

(11.5) — no `swap()` function throws an exception.

(11.6) — no `swap()` function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. [*Note*: The `end()` iterator does not refer to any element, so it may be invalidated. — *end note*]

12 Unless otherwise specified (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to a library function shall not invalidate iterators to, or change the values of, objects within that container.

13 A *contiguous container* is a container whose member types `iterator` and `const_iterator` meet the *Cpp17RandomAccessIterator* requirements (**??**) and model `ContiguousIterator` (**??**).

14 Table 64 lists operations that are provided for some types of containers but not others. Those containers for which the listed operations are provided shall implement the semantics described in Table 64 unless otherwise stated. If the iterators passed to `lexicographical_compare` satisfy the constexpr iterator requirements (**??**) then the operations described in Table 64 are implemented by constexpr functions.

Table 64 — Optional container operations

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition	Complexity
`a < b`	convertible to `bool`	`lexicographical_compare( a.begin(), a.end(), b.begin(), b.end())`	~~*Requires:*~~ *Expects:* `<` is defined for values of type (possibly `const`) `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

[*Note*: The algorithm `lexicographical_compare()` is defined in **??**. — *end note*]

15 All of the containers defined in this Clause and in **??** except `array` meet the additional requirements of an allocator-aware container, as described in Table 65.

Given an allocator type `A` and given a container type `X` having a `value_type` identical to `T` and an `allocator_type` identical to `allocator_traits<A>::rebind_alloc<T>` and given an lvalue `m` of type `A`, a pointer `p` of type `T*`, an expression `v` of type (possibly `const`) `T`, and an rvalue `rv` of type `T`, the following terms are defined. If `X` is not allocator-aware, the terms below are defined as if `A` were `allocator<T>` — no allocator object needs to be created and user specializations of `allocator<T>` are not instantiated:

(15.1) — `T` is *Cpp17DefaultInsertable into `X`* means that the following expression is well-formed:

```
allocator_traits<A>::construct(m, p)
```

(15.2) — An element of `X` is *default-inserted* if it is initialized by evaluation of the expression

```
allocator_traits<A>::construct(m, p)
```

where `p` is the address of the uninitialized storage for the element allocated within `X`.

(15.3) — `T` is *Cpp17MoveInsertable into `X`* means that the following expression is well-formed:

```
allocator_traits<A>::construct(m, p, rv)
```

and its evaluation causes the following postcondition to hold: The value of `*p` is equivalent to the value of `rv` before the evaluation. [*Note*: `rv` remains a valid object. Its state is unspecified — *end note*]

(15.4) — `T` is *Cpp17CopyInsertable into `X`* means that, in addition to `T` being *Cpp17MoveInsertable* into `X`, the following expression is well-formed:

```
allocator_traits<A>::construct(m, p, v)
```

and its evaluation causes the following postcondition to hold: The value of `v` is unchanged and is equivalent to `*p`.

(15.5) — `T` is *Cpp17EmplaceConstructible into `X` from `args`*, for zero or more arguments `args`, means that the following expression is well-formed:

```
allocator_traits<A>::construct(m, p, args)
```

(15.6) — `T` is *Cpp17Erasable from `X`* means that the following expression is well-formed:

```
allocator_traits<A>::destroy(m, p)
```

[*Note*: A container calls `allocator_traits<A>::construct(m, p, args)` to construct an element at `p` using `args`, with `m == get_allocator()`. The default `construct` in `allocator` will call `::new((void*)p) T(args)`, but specialized allocators may choose a different definition. — *end note*]

16 In Table 65, `X` denotes an allocator-aware container class with a `value_type` of `T` using allocator of type `A`, `u` denotes a variable, `a` and `b` denote non-const lvalues of type `X`, `t` denotes an lvalue or a const rvalue of type `X`, `rv` denotes a non-const rvalue of type `X`, and `m` is a value of type `A`.

Table 65 — Allocator-aware container requirements

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`allocator_type`	`A`	~~*Requires:*~~ *Mandates:* `allocator_type::value_type` is the same as `X::value_type`.	compile time
`get_allocator()`	`A`		constant
`X()` `X u;`		~~*Requires:*~~ *Expects:* `A` ~~is~~meets the *Cpp17DefaultConstructible* requirements. *Ensures:* `u.empty()` returns `true`, `u.get_allocator() == A()`	constant

`X(m)`
 	
`X u(m);`
		Ensures: `u.empty()` returns `true`,
		`u.get_allocator() == m`
			constant

`X(t, m)` `X u(t, m);`		~~*Requires:*~~ *Expects:* `T` is *Cpp17CopyInsertable* into `X`. *Ensures:* `u == t`, `u.get_allocator() == m`	linear
`X(rv)` `X u(rv);`		*Ensures:* `u` ~~shall have~~has the same elements as `rv` had before this construction; the value of `u.get_allocator()` ~~shall be~~is the same as the value of `rv.get_allocator()` before this construction.	constant
`X(rv, m)` `X u(rv, m);`		~~*Requires:*~~ *Expects:* `T` is *Cpp17MoveInsertable* into `X`. *Ensures:* `u` ~~shall have~~has the same elements, or copies of the elements, that `rv` had before this construction, `u.get_allocator() == m`	constant if `m == rv.get_allocator()`, otherwise linear
`a = t`	`X&`	~~*Requires:*~~ *Expects:* `T` is *Cpp17CopyInsertable* into `X` and *Cpp17CopyAssignable*. *Ensures:* `a == t`	linear

Table 65 — Allocator-aware container requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a = rv`	`X&`	~~*Requires:*~~ *Expects:* If `allocator_traits<allocator_type>::propagate_on_container_move_assignment::value` is `false`, `T` is *Cpp17MoveInsertable* into `X` and *Cpp17MoveAssignable*. [Editor's note: Add linebreak here.] All existing elements of `a` are either move assigned to or destroyed. *Ensures:* `a` shall be equal to the value that `rv` had before this assignment.	linear
`a.swap(b)`	`void`	exchanges the contents of `a` and `b`	constant

17 The behavior of certain container member functions and deduction guides depends on whether types qualify as input iterators or allocators. The extent to which an implementation determines that a type cannot be an input iterator is unspecified, except that as a minimum integral types shall not qualify as input iterators. Likewise, the extent to which an implementation determines that a type cannot be an allocator is unspecified, except that as a minimum a type `A` shall not qualify as an allocator unless it satisfies both of the following conditions:

(17.1) — The *qualified-id* `A::value_type` is valid and denotes a type (**??**).

(17.2) — The expression `declval<A&>().allocate(size_t{})` is well-formed when treated as an unevaluated operand.

21.2.2 Container data races [container.requirements.dataraces]

1 For purposes of avoiding data races (**??**), implementations shall consider the following functions to be `const`: `begin`, `end`, `rbegin`, `rend`, `front`, `back`, `data`, `find`, `lower_bound`, `upper_bound`, `equal_range`, `at` and, except in associative or unordered associative containers, `operator[]`.

2 Notwithstanding **??**, implementations are required to avoid data races when the contents of the contained object in different elements in the same container, excepting `vector<bool>`, are modified concurrently.

3 [*Note*: For a `vector<int> x` with a size greater than one, `x[1] = 5` and `*x.begin() = 10` can be executed concurrently without a data race, but `x[0] = 5` and `*x.begin() = 10` executed concurrently may result in a data race. As an exception to the general rule, for a `vector<bool> y`, `y[0] = true` may race with `y[1] = true`. — *end note*]

21.2.3 Sequence containers [sequence.reqmts]

1 A sequence container organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides four basic kinds of sequence containers: `vector`, `forward_list`, `list`, and `deque`. In addition, `array` is provided as a sequence container which provides limited sequence operations because it has a fixed number of elements. The library also provides container adaptors that make it easy to construct abstract data types, such as `stacks` or `queues`, out of the basic sequence container kinds (or out of other kinds of sequence containers that the user might define).

2 [*Note*: The sequence containers offer the programmer different complexity trade-offs and should be used accordingly. `vector` is the type of sequence container that should be used by default. `array` should be used when the container has a fixed size known during translation. `list` or `forward_list` should be used when there are frequent insertions and deletions from the middle of the sequence. `deque` is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence. When

choosing a container, remember `vector` is best; leave a comment to explain if you choose from the rest! — *end note*]

3 In Tables 66 and 67, `X` denotes a sequence container class, `a` denotes a value of type `X` containing elements of type `T`, `u` denotes the name of a variable being declared, `A` denotes `X::allocator_type` if the *qualified-id* `X::allocator_type` is valid and denotes a type (**??**) and `allocator<T>` if it doesn't, `i` and `j` denote iterators satisfying input iterator requirements and refer to elements implicitly convertible to `value_type`, `[i, j)` denotes a valid range, `il` designates an object of type `initializer_list<value_type>`, `n` denotes a value of type `X::size_type`, `p` denotes a valid constant iterator to `a`, `q` denotes a valid dereferenceable constant iterator to `a`, `[q1, q2)` denotes a valid range of constant iterators in `a`, `t` denotes an lvalue or a const rvalue of `X::value_type`, and `rv` denotes a non-const rvalue of `X::value_type`. `Args` denotes a template parameter pack; `args` denotes a function parameter pack with the pattern `Args&&`.

4 The complexities of the expressions are sequence dependent.

Table 66 — Sequence container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition
`X(n, t)` `X u(n, t);`		~~*Requires:*~~ *Expects:* `T` shall be *Cpp17CopyInsertable* into `X`. *Ensures:* `distance(begin(), end()) == n` Constructs a sequence container with `n` copies of `t`
`X(i, j)` `X u(i, j);`		~~*Requires:*~~ *Expects:* `T` shall be *Cpp17EmplaceConstructible* into `X` from `*i`. For `vector`, if the iterator does not meet the *Cpp17ForwardIterator* requirements (**??**), `T` shall also be *Cpp17MoveInsertable* into `X`.[Editor's note: Add linebreak here.] Each iterator in the range `[i, j)` ~~shall be~~is dereferenced exactly once. *Ensures:* `distance(begin(), end()) == distance(i, j)` Constructs a sequence container equal to the range `[i, j)`
`X(il)`		Equivalent to `X(il.begin(), il.end())`
`a = il`	`X&`	~~*Requires:*~~ *Expects:* `T` is *Cpp17CopyInsertable* into `X` and *Cpp17CopyAssignable*. [Editor's note: Add linebreak here.] Assigns the range `[il.begin(), il.end())` into `a`. All existing elements of `a` are either assigned to or destroyed. *Returns:* `*this`.
`a.emplace(p, args)`	`iterator`	~~*Requires:*~~ *Expects:* `T` is *Cpp17EmplaceConstructible* into `X` from `args`. For `vector` and `deque`, `T` is also *Cpp17MoveInsertable* into `X` and *Cpp17MoveAssignable*. [Editor's note: Add linebreak here.] *Effects:* Inserts an object of type `T` constructed with `std::forward<Args>(args)...` before `p`. [*Note*: `args` may directly or indirectly refer to a value in `a`. — *end note*]
`a.insert(p,t)`	`iterator`	~~*Requires:*~~ *Expects:* `T` shall be *Cpp17CopyInsertable* into `X`. For `vector` and `deque`, `T` shall also be *Cpp17CopyAssignable*. *Effects:* Inserts a copy of `t` before `p`.

Table 66 — Sequence container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition
`a.insert(p,rv)`	`iterator`	~~*Requires:*~~ *Expects:* `T` shall be *Cpp17MoveInsertable* into `X`. For `vector` and `deque`, `T` shall also be *Cpp17MoveAssignable*. *Effects:* Inserts a copy of `rv` before `p`.
`a.insert(p,n,t)`	`iterator`	~~*Requires:*~~ *Expects:* `T` shall be *Cpp17CopyInsertable* into `X` and *Cpp17CopyAssignable*. Inserts `n` copies of `t` before `p`.
`a.insert(p,i,j)`	`iterator`	~~*Requires:*~~ *Expects:* `T` shall be *Cpp17EmplaceConstructible* into `X` from `*i`. For `vector` and `deque`, `T` shall also be *Cpp17MoveInsertable* into `X`, *Cpp17MoveConstructible*, *Cpp17MoveAssignable*, and swappable (**??**). `i` and `j` are not iterators into `a`.[Editor's note: Add linebreak here.] Each iterator in the range `[i, j)` ~~shall be~~is dereferenced exactly once. ~~*Requires: i and j are not iterators into a.*~~ Inserts copies of elements in `[i, j)` before `p`
`a.insert(p, il)`	`iterator`	`a.insert(p, il.begin(), il.end())`.
`a.erase(q)`	`iterator`	~~*Requires:*~~ *Expects:* For `vector` and `deque`, `T` shall be *Cpp17MoveAssignable*. *Effects:* Erases the element pointed to by `q`.
`a.erase(q1,q2)`	`iterator`	~~*Requires:*~~ *Expects:* For `vector` and `deque`, `T` shall be *Cpp17MoveAssignable*. *Effects:* Erases the elements in the range `[q1, q2)`.
`a.clear()`	`void`	Destroys all elements in `a`. Invalidates all references, pointers, and iterators referring to the elements of `a` and may invalidate the past-the-end iterator. *Ensures:* `a.empty()` returns `true`. *Complexity:* Linear.
`a.assign(i,j)`	`void`	~~*Requires:*~~ *Expects:* `T` shall be *Cpp17EmplaceConstructible* into `X` from `*i` and assignable from `*i`. For `vector`, if the iterator does not meet the forward iterator requirements (**??**), `T` shall also be *Cpp17MoveInsertable* into `X`. `i` and `j` are not iterators into `a`.[Editor's note: Add linebreak here.] Each iterator in the range `[i, j)` ~~shall be~~is dereferenced exactly once. ~~*Requires: i, j are not iterators into a.*~~ Replaces elements in `a` with a copy of `[i, j)`. Invalidates all references, pointers and iterators referring to the elements of `a`. For `vector` and `deque`, also invalidates the past-the-end iterator.
`a.assign(il)`	`void`	`a.assign(il.begin(), il.end())`.

Table 66 — Sequence container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition
`a.assign(n,t)`	`void`	~~*Requires:*~~ *Expects:* `T` shall be *Cpp17CopyInsertable* into `X` and *Cpp17CopyAssignable*. [Editor's note: Remove line break here.] ~~*Requires:*~~ `t` is not a reference into `a`. Replaces elements in `a` with `n` copies of `t`. Invalidates all references, pointers and iterators referring to the elements of `a`. For `vector` and `deque`, also invalidates the past-the-end iterator.

5 The iterator returned from `a.insert(p, t)` points to the copy of `t` inserted into `a`.

6 The iterator returned from `a.insert(p, rv)` points to the copy of `rv` inserted into `a`.

7 The iterator returned from `a.insert(p, n, t)` points to the copy of the first element inserted into `a`, or `p` if `n == 0`.

8 The iterator returned from `a.insert(p, i, j)` points to the copy of the first element inserted into `a`, or `p` if `i == j`.

9 The iterator returned from `a.insert(p, il)` points to the copy of the first element inserted into `a`, or `p` if `il` is empty.

10 The iterator returned from `a.emplace(p, args)` points to the new element constructed from `args` into `a`.

11 The iterator returned from `a.erase(q)` points to the element immediately following `q` prior to the element being erased. If no such element exists, `a.end()` is returned.

12 The iterator returned by `a.erase(q1, q2)` points to the element pointed to by `q2` prior to any elements being erased. If no such element exists, `a.end()` is returned.

13 For every sequence container defined in this Clause and in **??**:

(13.1) — If the constructor

```
template<class InputIterator>
  X(InputIterator first, InputIterator last,
    const allocator_type& alloc = allocator_type());
```

is called with a type `InputIterator` that does not qualify as an input iterator, then the constructor shall not participate in overload resolution.

(13.2) — If the member functions of the forms:

```
template<class InputIterator>
  return-type F(const_iterator p,
                InputIterator first, InputIterator last);       // such as insert

template<class InputIterator>
  return-type F(InputIterator first, InputIterator last);       // such as append, assign

template<class InputIterator>
  return-type F(const_iterator i1, const_iterator i2,
                InputIterator first, InputIterator last);       // such as replace
```

are called with a type `InputIterator` that does not qualify as an input iterator, then these functions shall not participate in overload resolution.

(13.3) — A deduction guide for a sequence container shall not participate in overload resolution if it has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter, or if it has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.

14 Table 67 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 67 — Optional sequence container operations

Expression	Return type	Operational semantics	Container
`a.front()`	`reference; const_reference` for constant `a`	`*a.begin()`	`basic_string`, `array`, `deque`, `forward_list`, `list`, `vector`
`a.back()`	`reference; const_reference` for constant `a`	`{ auto tmp = a.end(); --tmp; return *tmp; }`	`basic_string`, `array`, `deque`, `list`, `vector`
`a.emplace_front(args)`	`reference`	Prepends an object of type `T` constructed with `std::forward<Args>(args)...`. ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17EmplaceConstructible* into `X` from `args`. *Returns:* `a.front()`.	`deque`, `forward_list`, `list`
`a.emplace_back(args)`	`reference`	Appends an object of type `T` constructed with `std::forward<Args>(args)...`. ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17EmplaceConstructible* into `X` from `args`. For `vector`, `T` shall also be *Cpp17MoveInsertable* into `X`. *Returns:* `a.back()`.	`deque`, `list`, `vector`
`a.push_front(t)`	`void`	Prepends a copy of `t`. ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17CopyInsertable* into `X`.	`deque`, `forward_list`, `list`
`a.push_front(rv)`	`void`	Prepends a copy of `rv`. ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17MoveInsertable* into `X`.	`deque`, `forward_list`, `list`
`a.push_back(t)`	`void`	Appends a copy of `t`. ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17CopyInsertable* into `X`.	`basic_string`, `deque`, `list`, `vector`
`a.push_back(rv)`	`void`	Appends a copy of `rv`. ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17MoveInsertable* into `X`.	`basic_string`, `deque`, `list`, `vector`
`a.pop_front()`	`void`	Destroys the first element. ~~*Requires:*~~ *Expects:* `a.empty()` shall be `false`.	`deque`, `forward_list`, `list`
`a.pop_back()`	`void`	Destroys the last element. ~~*Requires:*~~ *Expects:* `a.empty()` shall be `false`.	`basic_string`, `deque`, `list`, `vector`
`a[n]`	`reference; const_reference` for constant `a`	`*(a.begin() + n)`	`basic_string`, `array`, `deque`, `vector`
`a.at(n)`	`reference; const_reference` for constant `a`	`*(a.begin() + n)`	`basic_string`, `array`, `deque`, `vector`

15 The member function `at()` provides bounds-checked access to container elements. `at()` throws `out_of_range` if `n >= a.size()`.

21.2.4 Node handles [container.node]

21.2.4.1 Overview [container.node.overview]

1 A *node handle* is an object that accepts ownership of a single element from an associative container (21.2.6) or an unordered associative container (21.2.7). It may be used to transfer that ownership to another container with compatible nodes. Containers with compatible nodes have the same node handle type. Elements may be transferred in either direction between container types in the same row of Table 68.

Table 68 — Container types with compatible nodes

`map<K, T, C1, A>`	`map<K, T, C2, A>`
`map<K, T, C1, A>`	`multimap<K, T, C2, A>`
`set<K, C1, A>`	`set<K, C2, A>`
`set<K, C1, A>`	`multiset<K, C2, A>`
`unordered_map<K, T, H1, E1, A>`	`unordered_map<K, T, H2, E2, A>`
`unordered_map<K, T, H1, E1, A>`	`unordered_multimap<K, T, H2, E2, A>`
`unordered_set<K, H1, E1, A>`	`unordered_set<K, H2, E2, A>`
`unordered_set<K, H1, E1, A>`	`unordered_multiset<K, H2, E2, A>`

2 If a node handle is not empty, then it contains an allocator that is equal to the allocator of the container when the element was extracted. If a node handle is empty, it contains no allocator.

3 Class *`node-handle`* is for exposition only.

4 If a user-defined specialization of `pair` exists for `pair<const Key, T>` or `pair<Key, T>`, where `Key` is the container's `key_type` and `T` is the container's `mapped_type`, the behavior of operations involving node handles is undefined.

```
template<unspecified>
class node-handle {
public:
  // These type declarations are described in Tables 69 and 70.
  using value_type     = see below;      // not present for map containers
  using key_type       = see below;      // not present for set containers
  using mapped_type    = see below;      // not present for set containers
  using allocator_type = see below;

private:
  using container_node_type = unspecified;
  using ator_traits = allocator_traits<allocator_type>;

  typename ator_traits::rebind_traits<container_node_type>::pointer ptr_;
  optional<allocator_type> alloc_;

public:
  // 21.2.4.2, constructors, copy, and assignment
  constexpr node-handle() noexcept : ptr_(), alloc_() {}
  node-handle(node-handle&&) noexcept;
  node-handle& operator=(node-handle&&);

  // 21.2.4.3, destructor
  ~node-handle();

  // 21.2.4.4, observers
  value_type& value() const;            // not present for map containers
  key_type& key() const;                // not present for set containers
  mapped_type& mapped() const;          // not present for set containers

  allocator_type get_allocator() const;
  explicit operator bool() const noexcept;
  [[nodiscard]] bool empty() const noexcept;
```

```
  // 21.2.4.5, modifiers
  void swap(node-handle&)
    noexcept(ator_traits::propagate_on_container_swap::value ||
             ator_traits::is_always_equal::value);

  friend void swap(node-handle& x, node-handle& y) noexcept(noexcept(x.swap(y))) {
    x.swap(y);
  }
};
```

21.2.4.2 Constructors, copy, and assignment [container.node.cons]

```
node-handle(node-handle&& nh) noexcept;
```

1 *Effects:* Constructs a *node-handle* object initializing `ptr_` with `nh.ptr_`. Move constructs `alloc_` with `nh.alloc_`. Assigns `nullptr` to `nh.ptr_` and assigns `nullopt` to `nh.alloc_`.

```
node-handle& operator=(node-handle&& nh);
```

2 ~~*Requires:*~~ *Expects:* Either `!alloc_`, or `ator_traits::propagate_on_container_move_assignment::value` is `true`, or `alloc_ == nh.alloc_`.

3 *Effects:*

(3.1) — If `ptr_ != nullptr`, destroys the `value_type` subobject in the `container_node_type` object pointed to by `ptr_` by calling `ator_traits::destroy`, then deallocates `ptr_` by calling `ator_traits::rebind_traits<container_node_type>::deallocate`.

(3.2) — Assigns `nh.ptr_` to `ptr_`.

(3.3) — If `!alloc_` or `ator_traits::propagate_on_container_move_assignment::value` is `true`, move assigns `nh.alloc_` to `alloc_`.

(3.4) — Assigns `nullptr` to `nh.ptr_` and assigns `nullopt` to `nh.alloc_`.

4 *Returns:* `*this`.

5 *Throws:* Nothing.

21.2.4.3 Destructor [container.node.dtor]

```
~node-handle();
```

1 *Effects:* If `ptr_ != nullptr`, destroys the `value_type` subobject in the `container_node_type` object pointed to by `ptr_` by calling `ator_traits::destroy`, then deallocates `ptr_` by calling `ator_traits::rebind_traits<container_node_type>::deallocate`.

21.2.4.4 Observers [container.node.observers]

```
value_type& value() const;
```

1 ~~*Requires:*~~ *Expects:* `empty() == false`.

2 *Returns:* A reference to the `value_type` subobject in the `container_node_type` object pointed to by `ptr_`.

3 *Throws:* Nothing.

```
key_type& key() const;
```

4 ~~*Requires:*~~ *Expects:* `empty() == false`.

5 *Returns:* A non-const reference to the `key_type` member of the `value_type` subobject in the `container_node_type` object pointed to by `ptr_`.

6 *Throws:* Nothing.

7 *Remarks:* Modifying the key through the returned reference is permitted.

```
mapped_type& mapped() const;
```

8 ~~*Requires:*~~ *Expects:* `empty() == false`.

9 *Returns:* A reference to the `mapped_type` member of the `value_type` subobject in the `container_node_type` object pointed to by `ptr_`.

10 *Throws:* Nothing.

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

11 ~~*Requires:*~~ *Expects:* `empty() == false`.

12 *Returns:* `*alloc_`.

13 *Throws:* Nothing.

```
explicit operator bool() const noexcept;
```

14 *Returns:* `ptr_ != nullptr`.

```
[[nodiscard]] bool empty() const noexcept;
```

15 *Returns:* `ptr_ == nullptr`.

21.2.4.5 Modifiers [container.node.modifiers]

```
void swap(node-handle& nh)
  noexcept(ator_traits::propagate_on_container_swap::value ||
           ator_traits::is_always_equal::value);
```

1 ~~*Requires:*~~ *Expects:* `!alloc_`, or `!nh.alloc_`, or `ator_traits::propagate_on_container_swap::value` is `true`, or `alloc_ == nh.alloc_`.

2 *Effects:* Calls `swap(ptr_, nh.ptr_)`. If `!alloc_`, or `!nh.alloc_`, or `ator_traits::propagate_on_container_swap::value` is `true` calls `swap(alloc_, nh.alloc_)`.

21.2.5 Insert return type [container.insert.return]

1 The associative containers with unique keys and the unordered containers with unique keys have a member function `insert` that returns a nested type `insert_return_type`. That return type is a specialization of the template specified in this subclause.

```
template<class Iterator, class NodeType>
struct insert-return-type
{
  Iterator position;
  bool     inserted;
  NodeType node;
};
```

2 The name *`insert-return-type`* is exposition only. *`insert-return-type`* has the template parameters, data members, and special members specified above. It has no base classes or members other than those specified.

21.2.6 Associative containers [associative.reqmts]

1 Associative containers provide fast retrieval of data based on keys. The library provides four basic kinds of associative containers: `set`, `multiset`, `map` and `multimap`.

2 Each associative container is parameterized on `Key` and an ordering relation `Compare` that induces a strict weak ordering (**??**) on elements of `Key`. In addition, `map` and `multimap` associate an arbitrary *mapped type* `T` with the `Key`. The object of type `Compare` is called the *comparison object* of a container.

3 The phrase "equivalence of keys" means the equivalence relation imposed by the comparison object. That is, two keys `k1` and `k2` are considered to be equivalent if for the comparison object `comp`, `comp(k1, k2) == false && comp(k2, k1) == false`. [*Note*: This is not necessarily the same as the result of `k1 == k2`. — *end note*] For any two keys `k1` and `k2` in the same container, calling `comp(k1, k2)` shall always return the same value.

4 An associative container supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equivalent keys*. The `set` and `map` classes support unique keys; the `multiset` and `multimap` classes support equivalent keys. For `multiset` and `multimap`, `insert`, `emplace`, and `erase` preserve the relative ordering of equivalent elements.

5 For `set` and `multiset` the value type is the same as the key type. For `map` and `multimap` it is equal to `pair<const Key, T>`.

6 `iterator` of an associative container is of the bidirectional iterator category. 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. [*Note*: `iterator` and `const_iterator` have identical semantics in this case, and `iterator` is convertible to `const_iterator`. Users can avoid violating the one-definition rule by always using `const_iterator` in their function parameter lists. — *end note*]

7 The associative containers meet all the requirements of Allocator-aware containers (21.2.1), except that for `map` and `multimap`, the requirements placed on `value_type` in Table 65 apply instead to `key_type` and `mapped_type`. [*Note*: For example, in some cases `key_type` and `mapped_type` are required to be *Cpp17CopyAssignable* even though the associated `value_type`, `pair<const key_type, mapped_type>`, is not *Cpp17CopyAssignable*. — *end note*]

8 In Table 69, `X` denotes an associative container class, `a` denotes a value of type `X`, `a2` denotes a value of a type with nodes compatible with type `X` (Table 68), `b` denotes a possibly `const` value of type `X`, `u` denotes the name of a variable being declared, `a_uniq` denotes a value of type `X` when `X` supports unique keys, `a_eq` denotes a value of type `X` when `X` supports multiple keys, `a_tran` denotes a possibly `const` value of type `X` when the *qualified-id* `X::key_compare::is_transparent` is valid and denotes a type (**??**), `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 constant iterator to `a`, `q` denotes a valid dereferenceable constant iterator to `a`, `r` denotes a valid dereferenceable iterator to `a`, `[q1, q2)` denotes a valid range of constant iterators in `a`, `il` designates an object of type `initializer_list<value_type>`, `t` denotes a value of type `X::value_type`, `k` denotes a value of type `X::key_type` and `c` denotes a possibly `const` value of type `X::key_compare`; `kl` is a value such that `a` is partitioned (**??**) with respect to `c(r, kl)`, with `r` the key value of `e` and `e` in `a`; `ku` is a value such that `a` is partitioned with respect to `!c(ku, r)`; `ke` is a value such that `a` is partitioned with respect to `c(r, ke)` and `!c(ke, r)`, with `c(r, ke)` implying `!c(ke, r)`. `A` denotes the storage allocator used by `X`, if any, or `allocator<X::value_type>` otherwise, `m` denotes an allocator of a type convertible to `A`, and `nh` denotes a non-const rvalue of type `X::node_type`.

Table 69 — Associative container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::key_type`	`Key`		compile time
`X::mapped_type` (`map` and `multimap` only)	`T`		compile time
`X::value_type` (`set` and `multiset` only)	`Key`	~~*Requires:*~~ *Expects:* `value_type` is *Cpp17Erasable* from `X`	compile time
`X::value_type` (`map` and `multimap` only)	`pair<const Key, T>`	~~*Requires:*~~ *Expects:* `value_type` is *Cpp17Erasable* from `X`	compile time
`X::key_compare`	`Compare`	~~*Requires:*~~ *Expects:* `key_compare` is *Cpp17CopyConstructible*.	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.e., `Key`) for `map` and `multimap`.	compile time

Table 69 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::node_type`	a specialization of a *`node-handle`* class template, such that the public nested types are the same types as the corresponding types in `X`.	see 21.2.4	compile time
`X(c)` `X u(c);`		*Effects:* Constructs an empty container. Uses a copy of `c` as a comparison object.	constant
`X()` `X u;`		~~*Requires:*~~ *Expects:* `key_compare` ~~is~~meets the *Cpp17DefaultConstructible* requirements. *Effects:* Constructs an empty container. Uses `Compare()` as a comparison object	constant
`X(i,j,c)` `X u(i,j,c);`		~~*Requires:*~~ *Expects:* `value_type` is *Cpp17EmplaceConstructible* into `X` from `*i`. *Effects:* 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, where N has the value `distance(i, j)`; linear if `[i, j)` is sorted with `value_comp()`
`X(i,j)` `X u(i,j);`		~~*Requires:*~~ *Expects:* `key_compare` ~~is~~meets the *Cpp17DefaultConstructible* requirements. `value_type` is *Cpp17EmplaceConstructible* into `X` from `*i`. *Effects:* Same as above, but uses `Compare()` as a comparison object.	same as above
`X(il)`		same as `X(il.begin(), il.end())`	same as `X(il.begin(), il.end())`
`X(il,c)`		same as `X(il.begin(), il.end(), c)`	same as `X(il.begin(), il.end(), c)`
`a = il`	`X&`	~~*Requires:*~~ *Expects:* `value_type` is *Cpp17CopyInsertable* into `X` and *Cpp17CopyAssignable*. *Effects:* Assigns the range `[il.begin(), il.end())` into `a`. All existing elements of `a` are either assigned to or destroyed.	$N \log N$ in general, where N has the value `il.size() + a.size()`; linear if `[il.begin(), il.end())` is sorted with `value_comp()`
`b.key_comp()`	`X::key_compare`	returns the comparison object out of which `b` was constructed.	constant

Table 69 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`b.value_comp()`	`X::value_compare`	returns an object of `value_compare` constructed out of the comparison object	constant
`a_uniq.emplace(args)`	`pair<iterator, bool>`	~~*Requires:*~~ *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `X` from `args`. *Effects:* Inserts a `value_type` object `t` constructed with `std::forward<Args>(args)...` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair is `true` if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of `t`.	logarithmic
`a_eq.emplace(args)`	`iterator`	~~*Requires:*~~ *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `X` from `args`. *Effects:* Inserts a `value_type` object `t` constructed with `std::forward<Args>(args)...` and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to `t` exists in `a_eq`, `t` is inserted at the end of that range.	logarithmic
`a.emplace_hint(p, args)`	`iterator`	equivalent to `a.emplace(std::forward<Args>(args)...)`. Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The element is inserted as close as possible to the position just prior to `p`.	logarithmic in general, but amortized constant if the element is inserted right before `p`

Table 69 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a_uniq.insert(t)`	`pair<iterator, bool>`	~~*Requires:*~~ *Expects:* If `t` is a non-const rvalue, `value_type` shall be *Cpp17MoveInsertable* into `X`; otherwise, `value_type` shall be *Cpp17CopyInsertable* into `X`. *Effects:* Inserts `t` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair is `true` if and only if the insertion takes place, and the `iterator` component of the pair points to the element with key equivalent to the key of `t`.	logarithmic
`a_eq.insert(t)`	`iterator`	~~*Requires:*~~ *Expects:* If `t` is a non-const rvalue, `value_type` shall be *Cpp17MoveInsertable* into `X`; otherwise, `value_type` shall be *Cpp17CopyInsertable* into `X`. *Effects:* Inserts `t` and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to `t` exists in `a_eq`, `t` is inserted at the end of that range.	logarithmic
`a.insert(p, t)`	`iterator`	~~*Requires:*~~ *Expects:* If `t` is a non-const rvalue, `value_type` shall be *Cpp17MoveInsertable* into `X`; otherwise, `value_type` shall be *Cpp17CopyInsertable* into `X`. *Effects:* Inserts `t` if and only if there is no element with key equivalent to the key of `t` in containers with unique keys; always inserts `t` in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to the key of `t`. `t` is inserted as close as possible to the position just prior to `p`.	logarithmic in general, but amortized constant if `t` is inserted right before `p`.

Table 69 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.insert(i, j)`	`void`	~~*Requires:*~~ *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `X` from `*i`. [Editor's note: Remove line break here.] ~~*Requires:*~~ `i`~~,~~ and `j` are not iterators into `a`. [Editor's note: Add line break here.] inserts each element from the range `[i, j)` if and only if there is no element with key equivalent to the key of that element in containers with unique keys; always inserts that element in containers with equivalent keys.	$N \log(\texttt{a.size()} + N)$, where N has the value `distance(i, j)`
`a.insert(il)`	`void`	equivalent to `a.insert(il.begin(), il.end())`	
`a_uniq.insert(nh)`	`insert_return_type`	~~*Requires:*~~ *Expects:* `nh` is empty or `a_uniq.get_allocator() == nh.get_allocator()`. *Effects:* If `nh` is empty, has no effect. Otherwise, inserts the element owned by `nh` if and only if there is no element in the container with a key equivalent to `nh.key()`. *Ensures:* If `nh` is empty, `inserted` is `false`, `position` is `end()`, and `node` is empty. Otherwise if the insertion took place, `inserted` is `true`, `position` points to the inserted element, and `node` is empty; if the insertion failed, `inserted` is `false`, `node` has the previous value of `nh`, and `position` points to an element with a key equivalent to `nh.key()`.	logarithmic

Table 69 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a_eq.insert(nh)`	`iterator`	~~*Requires:*~~ *Expects:* `nh` is empty or `a_eq.get_allocator() == nh.get_allocator()`. *Effects:* If `nh` is empty, has no effect and returns `a_eq.end()`. Otherwise, inserts the element owned by `nh` and returns an iterator pointing to the newly inserted element. If a range containing elements with keys equivalent to `nh.key()` exists in `a_eq`, the element is inserted at the end of that range. *Ensures:* `nh` is empty.	logarithmic
`a.insert(p, nh)`	`iterator`	~~*Requires:*~~ *Expects:* `nh` is empty or `a.get_allocator() == nh.get_allocator()`. *Effects:* If `nh` is empty, has no effect and returns `a.end()`. Otherwise, inserts the element owned by `nh` if and only if there is no element with key equivalent to `nh.key()` in containers with unique keys; always inserts the element owned by `nh` in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to `nh.key()`. The element is inserted as close as possible to the position just prior to `p`. *Ensures:* `nh` is empty if insertion succeeds, unchanged if insertion fails.	logarithmic in general, but amortized constant if the element is inserted right before `p`.
`a.extract(k)`	`node_type`	removes the first element in the container with key equivalent to `k`. Returns a `node_type` owning the element if found, otherwise an empty `node_type`.	log(`a.size()`)
`a.extract(q)`	`node_type`	removes the element pointed to by `q`. Returns a `node_type` owning that element.	amortized constant

Table 69 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.merge(a2)`	`void`	~~*Requires:*~~ *Expects:* `a.get_allocator() == a2.get_allocator()`. Attempts to extract each element in `a2` and insert it into `a` using the comparison object of `a`. In containers with unique keys, if there is an element in `a` with key equivalent to the key of an element from `a2`, then that element is not extracted from `a2`. *Ensures:* Pointers and references to the transferred elements of `a2` refer to those same elements but as members of `a`. Iterators referring to the transferred elements will continue to refer to their elements, but they now behave as iterators into `a`, not into `a2`. *Throws:* Nothing unless the comparison object throws.	$N \log($`a.size()`$+N)$, where N has the value `a2.size()`.
`a.erase(k)`	`size_type`	erases all elements in the container with key equivalent to `k`. returns the number of erased elements.	$\log($`a.size()`$) +$ `a.count(k)`
`a.erase(q)`	`iterator`	erases the element pointed to by `q`. Returns an iterator pointing to the element immediately following `q` prior to the element being erased. If no such element exists, returns `a.end()`.	amortized constant
`a.erase(r)`	`iterator`	erases the element pointed to by `r`. Returns an iterator pointing to the element immediately following `r` prior to the element being erased. If no such element exists, returns `a.end()`.	amortized constant
`a.erase( q1, q2)`	`iterator`	erases all the elements in the range `[q1, q2)`. Returns an iterator pointing to the element pointed to by `q2` prior to any elements being erased. If no such element exists, `a.end()` is returned.	$\log($`a.size()`$) + N$, where N has the value `distance(q1, q2)`.
`a.clear()`	`void`	`a.erase(a.begin(),a.end())` *Ensures:* `a.empty()` returns `true`.	linear in `a.size()`.

Table 69 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`b.find(k)`	`iterator`; `const_iterator` for constant `b`.	returns an iterator pointing to an element with the key equivalent to `k`, or `b.end()` if such an element is not found	logarithmic
`a_tran.find(ke)`	`iterator`; `const_iterator` for constant `a_tran`.	returns an iterator pointing to an element with key `r` such that `!c(r, ke) && !c(ke, r)`, or `a_tran.end()` if such an element is not found	logarithmic
`b.count(k)`	`size_type`	returns the number of elements with key equivalent to `k`	log(`b.size()`) + `b.count(k)`
`a_tran.count(ke)`	`size_type`	returns the number of elements with key `r` such that `!c(r, ke) && !c(ke, r)`	log(`a_tran.size()`) + `a_tran.count(ke)`
`b.contains(k)`	`bool`	equivalent to `b.find(k) != b.end()`	logarithmic
`a_tran.contains(ke)`	`bool`	equivalent to `a_tran.find(ke) != a_tran.end()`	logarithmic
`b.lower_bound(k)`	`iterator`; `const_iterator` for constant `b`.	returns an iterator pointing to the first element with key not less than `k`, or `b.end()` if such an element is not found.	logarithmic
`a_tran.lower_bound(kl)`	`iterator`; `const_iterator` for constant `a_tran`.	returns an iterator pointing to the first element with key `r` such that `!c(r, kl)`, or `a_tran.end()` if such an element is not found.	logarithmic
`b.upper_bound(k)`	`iterator`; `const_iterator` for constant `b`.	returns an iterator pointing to the first element with key greater than `k`, or `b.end()` if such an element is not found.	logarithmic
`a_tran.upper_bound(ku)`	`iterator`; `const_iterator` for constant `a_tran`.	returns an iterator pointing to the first element with key `r` such that `c(ku, r)`, or `a_tran.end()` if such an element is not found.	logarithmic
`b.equal_range(k)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for constant `b`.	equivalent to `make_pair(b.lower_bound(k), b.upper_bound(k))`.	logarithmic

Table 69 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a_tran.equal_range(ke)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for constant `a_tran`.	equivalent to `make_pair(a_tran.lower_bound(ke), a_tran.upper_bound(ke))`.	logarithmic

9 The `insert` and `emplace` members shall not affect the validity of iterators and references to the container, and the `erase` members shall invalidate only iterators and references to the erased elements.

10 The `extract` members invalidate only iterators to the removed element; pointers and references to the removed element remain valid. However, accessing the element through such pointers and references while the element is owned by a `node_type` is undefined behavior. References and pointers to an element obtained while it is owned by a `node_type` are invalidated if the element is successfully inserted.

11 The fundamental property of iterators of associative containers is that they iterate through the containers in the non-descending order of keys where non-descending is defined by the comparison that was used to construct them. For any two dereferenceable iterators `i` and `j` such that distance from `i` to `j` is positive, the following condition holds:

```
value_comp(*j, *i) == false
```

12 For associative containers with unique keys the stronger condition holds:

```
value_comp(*i, *j) != false
```

13 When an associative container is constructed by passing a comparison object the container shall not store a pointer or reference to the passed object, even if that object is passed by reference. When an associative container is copied, either through a copy constructor or an assignment operator, the target container shall then use the comparison object from the container being copied, as if that comparison object had been passed to the target container in its constructor.

14 The member function templates `find`, `count`, `contains`, `lower_bound`, `upper_bound`, and `equal_range` shall not participate in overload resolution unless the *qualified-id* `Compare::is_transparent` is valid and denotes a type (**??**).

15 A deduction guide for an associative container shall not participate in overload resolution if any of the following are true:

(15.1) — It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.

(15.2) — It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.

(15.3) — It has a `Compare` template parameter and a type that qualifies as an allocator is deduced for that parameter.

21.2.6.1 Exception safety guarantees [associative.reqmts.except]

1 For associative containers, no `clear()` function throws an exception. `erase(k)` does not throw an exception unless that exception is thrown by the container's `Compare` object (if any).

2 For associative containers, if an exception is thrown by any operation from within an `insert` or `emplace` function inserting a single element, the insertion has no effect.

3 For associative containers, no `swap` function throws an exception unless that exception is thrown by the swap of the container's `Compare` object (if any).

21.2.7 Unordered associative containers [unord.req]

1 Unordered associative containers provide an ability for fast retrieval of data based on keys. The worst-case complexity for most operations is linear, but the average case is much faster. The library provides four unordered associative containers: `unordered_set`, `unordered_map`, `unordered_multiset`, and `unordered_multimap`.

2 Unordered associative containers conform to the requirements for Containers (21.2), except that the expressions `a == b` and `a != b` have different semantics than for the other container types.

3 Each unordered associative container is parameterized by `Key`, by a function object type `Hash` that meets the *Cpp17Hash* requirements (**??**) and acts as a hash function for argument values of type `Key`, and by a binary predicate `Pred` that induces an equivalence relation on values of type `Key`. Additionally, `unordered_map` and `unordered_multimap` associate an arbitrary *mapped type* `T` with the `Key`.

4 The container's object of type `Hash` — denoted by `hash` — is called the *hash function* of the container. The container's object of type `Pred` — denoted by `pred` — is called the *key equality predicate* of the container.

5 Two values `k1` and `k2` are considered equivalent if the container's key equality predicate `pred(k1, k2)` is valid and returns `true` when passed those values. If `k1` and `k2` are equivalent, the container's hash function shall return the same value for both. [*Note*: Thus, when an unordered associative container is instantiated with a non-default `Pred` parameter it usually needs a non-default `Hash` parameter as well. — *end note*] For any two keys `k1` and `k2` in the same container, calling `pred(k1, k2)` shall always return the same value. For any key `k` in a container, calling `hash(k)` shall always return the same value.

6 An unordered associative container supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equivalent keys*. `unordered_set` and `unordered_map` support unique keys. `unordered_multiset` and `unordered_multimap` support equivalent keys. In containers that support equivalent keys, elements with equivalent keys are adjacent to each other in the iteration order of the container. Thus, although the absolute order of elements in an unordered container is not specified, its elements are grouped into *equivalent-key groups* such that all elements of each group have equivalent keys. Mutating operations on unordered containers shall preserve the relative order of elements within each equivalent-key group unless otherwise specified.

7 For `unordered_set` and `unordered_multiset` the value type is the same as the key type. For `unordered_map` and `unordered_multimap` it is `pair<const Key, T>`.

8 For unordered 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. [*Note*: `iterator` and `const_iterator` have identical semantics in this case, and `iterator` is convertible to `const_iterator`. Users can avoid violating the one-definition rule by always using `const_iterator` in their function parameter lists. — *end note*]

9 The elements of an unordered associative container are organized into *buckets*. Keys with the same hash code appear in the same bucket. The number of buckets is automatically increased as elements are added to an unordered associative container, so that the average number of elements per bucket is kept below a bound. Rehashing invalidates iterators, changes ordering between elements, and changes which buckets elements appear in, but does not invalidate pointers or references to elements. For `unordered_multiset` and `unordered_multimap`, rehashing preserves the relative ordering of equivalent elements.

10 The unordered associative containers meet all the requirements of Allocator-aware containers (21.2.1), except that for `unordered_map` and `unordered_multimap`, the requirements placed on `value_type` in Table 65 apply instead to `key_type` and `mapped_type`. [*Note*: For example, `key_type` and `mapped_type` are sometimes required to be *Cpp17CopyAssignable* even though the associated `value_type`, `pair<const key_type, mapped_type>`, is not *Cpp17CopyAssignable*. — *end note*]

11 In Table 70:

(11.1) — `X` denotes an unordered associative container class,

(11.2) — `a` denotes a value of type `X`,

(11.3) — `a2` denotes a value of a type with nodes compatible with type `X` (Table 68),

(11.4) — `b` denotes a possibly const value of type `X`,

(11.5) — `a_uniq` denotes a value of type `X` when `X` supports unique keys,

(11.6) — `a_eq` denotes a value of type `X` when `X` supports equivalent keys,

(11.7) — `a_tran` denotes a possibly const value of type `X` when the *qualified-id* `X::hasher::transparent_key_equal` is valid and denotes a type (`??`),

(11.8) — `i` and `j` denote input iterators that refer to `value_type`,

(11.9) — `[i, j)` denotes a valid range,

(11.10) — `p` and `q2` denote valid constant iterators to `a`,

(11.11) — `q` and `q1` denote valid dereferenceable constant iterators to `a`,

(11.12) — `r` denotes a valid dereferenceable iterator to `a`,

(11.13) — `[q1, q2)` denotes a valid range in `a`,

(11.14) — `il` denotes a value of type `initializer_list<value_type>`,

(11.15) — `t` denotes a value of type `X::value_type`,

(11.16) — `k` denotes a value of type `key_type`,

(11.17) — `hf` denotes a possibly const value of type `hasher`,

(11.18) — `eq` denotes a possibly const value of type `key_equal`,

(11.19) — `ke` is a value such that

(11.19.1) — `eq(r1, ke) == eq(ke, r1)`

(11.19.2) — `hf(r1) == hf(ke)` if `eq(r1, ke)` is `true`, and

(11.19.3) — `(eq(r1, ke) && eq(r1, r2)) == eq(r2, ke)`

where `r1` and `r2` are keys of elements in `a_tran`,

(11.20) — `n` denotes a value of type `size_type`,

(11.21) — `z` denotes a value of type `float`, and

(11.22) — `nh` denotes a non-const rvalue of type `X::node_type`.

Table 70 — Unordered associative container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::key_type`	`Key`		compile time
`X::mapped_type` (`unordered_map` and `unordered_multimap` only)	`T`		compile time
`X::value_type` (`unordered_set` and `unordered_multiset` only)	`Key`	~~*Requires:*~~ *Expects:* `value_type` is *Cpp17Erasable* from `X`	compile time
`X::value_type` (`unordered_map` and `unordered_multimap` only)	`pair<const Key, T>`	~~*Requires:*~~ *Expects:* `value_type` is *Cpp17Erasable* from `X`	compile time
`X::hasher`	`Hash`	`Hash` shall be a unary function object type such that the expression `hf(k)` has type `size_t`.	compile time

Table 70 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::key_equal`	`Hash::transparent_key_equal` if such a *qualified-id* is valid and denotes a type (**??**); otherwise, `Pred`.	~~*Requires:*~~ *Expects:* `Pred` ~~is~~meets the *Cpp17CopyConstructible* requirements. `Pred` shall be a binary predicate that takes two arguments of type `Key`. `Pred` is an equivalence relation.	compile time
`X::local_iterator`	An iterator type whose category, value type, difference type, and pointer and reference types are the same as `X::iterator`'s.	A `local_iterator` object may be used to iterate through a single bucket, but may not be used to iterate across buckets.	compile time
`X::const_local_iterator`	An iterator type whose category, value type, difference type, and pointer and reference types are the same as `X::const_iterator`'s.	A `const_local_iterator` object may be used to iterate through a single bucket, but may not be used to iterate across buckets.	compile time
`X::node_type`	a specialization of a *`node-handle`* class template, such that the public nested types are the same types as the corresponding types in `X`.	see 21.2.4	compile time
`X(n, hf, eq)` `X a(n, hf, eq);`	`X`	*Effects:* Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `eq` as the key equality predicate.	$\mathcal{O}$(`n`)
`X(n, hf)` `X a(n, hf);`	`X`	~~*Requires:*~~ *Expects:* `key_equal` ~~is~~meets the *Cpp17DefaultConstructible* requirements. *Effects:* Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `key_equal()` as the key equality predicate.	$\mathcal{O}$(`n`)
`X(n)` `X a(n);`	`X`	~~*Requires:*~~ *Expects:* `hasher` and `key_equal` ~~are~~meet the *Cpp17DefaultConstructible* requirements. *Effects:* Constructs an empty container with at least `n` buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate.	$\mathcal{O}$(`n`)

Table 70 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X()` `X a;`	`X`	~~*Requires:*~~ *Expects:* `hasher` and `key_equal` ~~are~~meet the *Cpp17DefaultConstructible* requirements. *Effects:* Constructs an empty container with an unspecified number of buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate.	constant
`X(i, j, n, hf, eq)` `X a(i, j, n, hf, eq);`	`X`	~~*Requires:*~~ *Expects:* `value_type` is *Cpp17EmplaceConstructible* into `X` from `*i`. *Effects:* Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `eq` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case $\mathcal{O}(N)$ (N is `distance(i, j)`), worst case $\mathcal{O}(N^2)$
`X(i, j, n, hf)` `X a(i, j, n, hf);`	`X`	~~*Requires:*~~ *Expects:* `key_equal` ~~is~~meets the *Cpp17DefaultConstructible* requirements. `value_type` is *Cpp17EmplaceConstructible* into `X` from `*i`. *Effects:* Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `key_equal()` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case $\mathcal{O}(N)$ (N is `distance(i, j)`), worst case $\mathcal{O}(N^2)$
`X(i, j, n)` `X a(i, j, n);`	`X`	~~*Requires:*~~ *Expects:* `hasher` and `key_equal` ~~are~~meet the *Cpp17DefaultConstructible* requirements. `value_type` is *Cpp17EmplaceConstructible* into `X` from `*i`. *Effects:* Constructs an empty container with at least `n` buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case $\mathcal{O}(N)$ (N is `distance(i, j)`), worst case $\mathcal{O}(N^2)$

Table 70 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X(i, j)` `X a(i, j);`	`X`	~~*Requires:*~~ *Expects:* `hasher` and `key_equal` ~~are~~meet the *Cpp17DefaultConstructible* requirements. `value_type` is *Cpp17EmplaceConstructible* into `X` from `*i`. *Effects:* Constructs an empty container with an unspecified number of buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case $\mathcal{O}(N)$ (N is `distance(i, j)`), worst case $\mathcal{O}(N^2)$
`X(il)`	`X`	Same as `X(il.begin(), il.end())`.	Same as `X(il.begin(), il.end())`.
`X(il, n)`	`X`	Same as `X(il.begin(), il.end(), n)`.	Same as `X(il.begin(), il.end(), n)`.
`X(il, n, hf)`	`X`	Same as `X(il.begin(), il.end(), n, hf)`.	Same as `X(il.begin(), il.end(), n, hf)`.
`X(il, n, hf, eq)`	`X`	Same as `X(il.begin(), il.end(), n, hf, eq)`.	Same as `X(il.begin(), il.end(), n, hf, eq)`.
`X(b)` `X a(b);`	`X`	Copy constructor. In addition to the requirements of Table 62, copies the hash function, predicate, and maximum load factor.	Average case linear in `b.size()`, worst case quadratic.
`a = b`	`X&`	Copy assignment operator. In addition to the requirements of Table 62, copies the hash function, predicate, and maximum load factor.	Average case linear in `b.size()`, worst case quadratic.
`a = il`	`X&`	~~*Requires:*~~ *Expects:* `value_type` is *Cpp17CopyInsertable* into `X` and *Cpp17CopyAssignable*. *Effects:* Assigns the range `[il.begin(), il.end())` into `a`. All existing elements of `a` are either assigned to or destroyed.	Same as `a = X(il)`.
`b.hash_function()`	`hasher`	Returns `b`'s hash function.	constant
`b.key_eq()`	`key_equal`	Returns `b`'s key equality predicate.	constant

Table 70 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a_uniq.emplace(args)`	`pair<iterator, bool>`	~~*Requires:*~~ *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `X` from `args`. *Effects:* Inserts a `value_type` object `t` constructed with `std::forward<Args>(args)...` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair is `true` if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of `t`.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_uniq.size()`).
`a_eq.emplace(args)`	`iterator`	~~*Requires:*~~ *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `X` from `args`. *Effects:* Inserts a `value_type` object `t` constructed with `std::forward<Args>(args)...` and returns the iterator pointing to the newly inserted element.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_eq.size()`).
`a.emplace_hint(p, args)`	`iterator`	~~*Requires:*~~ *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `X` from `args`. *Effects:* Equivalent to `a.emplace( std::forward<Args>(args)...)`. Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The `const_iterator p` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).

Table 70 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a_uniq.insert(t)`	`pair<iterator, bool>`	~~*Requires:*~~ *Expects:* If `t` is a non-const rvalue, `value_type` shall be *Cpp17MoveInsertable* into `X`; otherwise, `value_type` shall be *Cpp17CopyInsertable* into `X`. *Effects:* Inserts `t` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair indicates whether the insertion takes place, and the `iterator` component points to the element with key equivalent to the key of `t`.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_uniq.size()`).
`a_eq.insert(t)`	`iterator`	~~*Requires:*~~ *Expects:* If `t` is a non-const rvalue, `value_type` shall be *Cpp17MoveInsertable* into `X`; otherwise, `value_type` shall be *Cpp17CopyInsertable* into `X`. *Effects:* Inserts `t`, and returns an iterator pointing to the newly inserted element.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_eq.size()`).
`a.insert(p, t)`	`iterator`	~~*Requires:*~~ *Expects:* If `t` is a non-const rvalue, `value_type` shall be *Cpp17MoveInsertable* into `X`; otherwise, `value_type` shall be *Cpp17CopyInsertable* into `X`. *Effects:* Equivalent to `a.insert(t)`. Return value is an iterator pointing to the element with the key equivalent to that of `t`. The iterator `p` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).
`a.insert(i, j)`	`void`	~~*Requires:*~~ *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `X` from `*i`. [Editor's note: Remove line break here.] ~~*Requires:*~~ `i` and `j` are not iterators in `a`. *Effects:* Equivalent to `a.insert(t)` for each element in `[i,j)`.	Average case $\mathcal{O}(N)$, where N is `distance(i, j)`. Worst case $\mathcal{O}(N$(`a.size()` $+1$)).

Table 70 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.insert(il)`	`void`	Same as `a.insert(il.begin(), il.end())`.	Same as `a.insert( il.begin(), il.end())`.
`a_uniq. insert(nh)`	`insert_return_type`	~~*Requires:*~~ *Expects:* `nh` is empty or `a_uniq.get_allocator() == nh.get_allocator()`. *Effects:* If `nh` is empty, has no effect. Otherwise, inserts the element owned by `nh` if and only if there is no element in the container with a key equivalent to `nh.key()`. *Ensures:* If `nh` is empty, `inserted` is `false`, `position` is `end()`, and `node` is empty. Otherwise if the insertion took place, `inserted` is `true`, `position` points to the inserted element, and `node` is empty; if the insertion failed, `inserted` is `false`, `node` has the previous value of `nh`, and `position` points to an element with a key equivalent to `nh.key()`.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_uniq. size()`).
`a_eq. insert(nh)`	`iterator`	~~*Requires:*~~ *Expects:* `nh` is empty or `a_eq.get_allocator() == nh.get_allocator()`. *Effects:* If `nh` is empty, has no effect and returns `a_eq.end()`. Otherwise, inserts the element owned by `nh` and returns an iterator pointing to the newly inserted element. *Ensures:* `nh` is empty.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_eq. size()`).

Table 70 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.insert(q, nh)`	`iterator`	~~*Requires:*~~ *Expects:* `nh` is empty or `a.get_allocator() == nh.get_allocator()`. *Effects:* If `nh` is empty, has no effect and returns `a.end()`. Otherwise, inserts the element owned by `nh` if and only if there is no element with key equivalent to `nh.key()` in containers with unique keys; always inserts the element owned by `nh` in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to `nh.key()`. The iterator `q` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint. *Ensures:* `nh` is empty if insertion succeeds, unchanged if insertion fails.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).
`a.extract(k)`	`node_type`	Removes an element in the container with key equivalent to `k`. Returns a `node_type` owning the element if found, otherwise an empty `node_type`.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).
`a.extract(q)`	`node_type`	Removes the element pointed to by `q`. Returns a `node_type` owning that element.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).

Table 70 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.merge(a2)`	`void`	~~*Requires:*~~ *Expects:* `a.get_allocator() == a2.get_allocator()`. Attempts to extract each element in `a2` and insert it into `a` using the hash function and key equality predicate of `a`. In containers with unique keys, if there is an element in `a` with key equivalent to the key of an element from `a2`, then that element is not extracted from `a2`. *Ensures:* Pointers and references to the transferred elements of `a2` refer to those same elements but as members of `a`. Iterators referring to the transferred elements and all iterators referring to `a` will be invalidated, but iterators to elements remaining in `a2` will remain valid.	Average case $\mathcal{O}(N)$, where N is `a2.size()`. Worst case $\mathcal{O}(N$`*a.size()+`$N)$.
`a.erase(k)`	`size_type`	Erases all elements with key equivalent to `k`. Returns the number of elements erased.	Average case $\mathcal{O}$(`a.count(k)`). Worst case $\mathcal{O}$(`a.size()`).
`a.erase(q)`	`iterator`	Erases the element pointed to by `q`. Returns the iterator immediately following `q` prior to the erasure.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).
`a.erase(r)`	`iterator`	Erases the element pointed to by `r`. Returns the iterator immediately following `r` prior to the erasure.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).
`a.erase(q1, q2)`	`iterator`	Erases all elements in the range `[q1, q2)`. Returns the iterator immediately following the erased elements prior to the erasure.	Average case linear in `distance(q1, q2)`, worst case $\mathcal{O}$(`a.size()`).
`a.clear()`	`void`	Erases all elements in the container. *Ensures:* `a.empty()` returns `true`	Linear in `a.size()`.
`b.find(k)`	`iterator`; `const_iterator` for const `b`.	Returns an iterator pointing to an element with key equivalent to `k`, or `b.end()` if no such element exists.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`b.size()`).
`a_tran.find(ke)`	`iterator`; `const_iterator` for const `a_tran`.	Returns an iterator pointing to an element with key equivalent to `ke`, or `a_tran.end()` if no such element exists.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_tran.size()`).

Table 70 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`b.count(k)`	`size_type`	Returns the number of elements with key equivalent to `k`.	Average case $\mathcal{O}$(`b.count(k)`), worst case $\mathcal{O}$(`b.size()`).
`a_tran.count(ke)`	`size_type`	Returns the number of elements with key equivalent to `ke`.	Average case $\mathcal{O}$(`a_tran.count(ke)`), worst case $\mathcal{O}$(`a_tran.size()`).
`b.contains(k)`	`bool`	Equivalent to `b.find(k) != b.end()`	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`b.size()`).
`a_tran.contains(ke)`	`bool`	Equivalent to `a_tran.find(ke) != a_tran.end()`	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_tran.size()`).
`b.equal_range(k)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for const `b`.	Returns a range containing all elements with keys equivalent to `k`. Returns `make_pair(b.end(), b.end())` if no such elements exist.	Average case $\mathcal{O}$(`b.count(k)`). Worst case $\mathcal{O}$(`b.size()`).
`a_tran.equal_range(ke)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for const `a_tran`.	Returns a range containing all elements with keys equivalent to `ke`. Returns `make_pair(a_tran.end(), a_tran.end())` if no such elements exist.	Average case $\mathcal{O}$(`a_tran.count(ke)`). Worst case $\mathcal{O}$(`a_tran.size()`).
`b.bucket_count()`	`size_type`	Returns the number of buckets that `b` contains.	Constant
`b.max_bucket_count()`	`size_type`	Returns an upper bound on the number of buckets that `b` might ever contain.	Constant
`b.bucket(k)`	`size_type`	~~*Requires:*~~ *Expects:* `b.bucket_count() > 0`. Returns the index of the bucket in which elements with keys equivalent to `k` would be found, if any such element existed. *Ensures:* the return value shall be in the range `[0, b.bucket_count())`.	Constant
`b.bucket_size(n)`	`size_type`	~~*Requires:*~~ *Expects:* `n` shall be in the range `[0, b.bucket_count())`.[Editor's note: Add line break here] Returns the number of elements in the `n`th bucket.	$\mathcal{O}$(`b.bucket_size(n)`)

Table 70 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`b.begin(n)`	`local_iterator`; `const_local_iterator` for const `b`.	~~*Requires:*~~ *Expects:* `n` shall be in the range `[0, b.bucket_count())`.[Editor's note: Add line break here] `b.begin(n)` returns an iterator referring to the first element in the bucket. If the bucket is empty, then `b.begin(n) == b.end(n)`.	Constant
`b.end(n)`	`local_iterator`; `const_local_iterator` for const `b`.	~~*Requires:*~~ *Expects:* `n` shall be in the range `[0, b.bucket_count())`.[Editor's note: Add line break here] `b.end(n)` returns an iterator which is the past-the-end value for the bucket.	Constant
`b.cbegin(n)`	`const_local_iterator`	~~*Requires:*~~ *Expects:* `n` shall be in the range `[0, b.bucket_count())`.[Editor's note: Add line break here] `b.cbegin(n)` returns an iterator referring to the first element in the bucket. If the bucket is empty, then `b.cbegin(n) == b.cend(n)`.	Constant
`b.cend(n)`	`const_local_iterator`	~~*Requires:*~~ *Expects:* `n` shall be in the range `[0, b.bucket_count())`.[Editor's note: Add line break here] `b.cend(n)` returns an iterator which is the past-the-end value for the bucket.	Constant
`b.load_factor()`	`float`	Returns the average number of elements per bucket.	Constant
`b.max_load_factor()`	`float`	Returns a positive number that the container attempts to keep the load factor less than or equal to. The container automatically increases the number of buckets as necessary to keep the load factor below this number.	Constant
`a.max_load_factor(z)`	`void`	~~*Requires:*~~ *Expects:* `z` shall be positive. May change the container's maximum load factor, using `z` as a hint.	Constant
`a.rehash(n)`	`void`	*Ensures:* `a.bucket_count() >= a.size() / a.max_load_factor()` and `a.bucket_count() >= n`.	Average case linear in `a.size()`, worst case quadratic.

Table 70 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.reserve(n)`	`void`	Same as `a.rehash(ceil(n / a.max_load_factor()))`.	Average case linear in `a.size()`, worst case quadratic.

12 Two unordered containers `a` and `b` compare equal if `a.size() == b.size()` and, for every equivalent-key group `[Ea1, Ea2)` obtained from `a.equal_range(Ea1)`, there exists an equivalent-key group `[Eb1, Eb2)` obtained from `b.equal_range(Ea1)`, such that `is_permutation(Ea1, Ea2, Eb1, Eb2)` returns `true`. For `unordered_set` and `unordered_map`, the complexity of `operator==` (i.e., the number of calls to the `==` operator of the `value_type`, to the predicate returned by `key_eq()`, and to the hasher returned by `hash_function()`) is proportional to N in the average case and to N^2 in the worst case, where N is a.size(). For `unordered_multiset` and `unordered_multimap`, the complexity of `operator==` is proportional to $\sum E_i^2$ in the average case and to N^2 in the worst case, where N is `a.size()`, and E_i is the size of the i^{th} equivalent-key group in `a`. However, if the respective elements of each corresponding pair of equivalent-key groups Ea_i and Eb_i are arranged in the same order (as is commonly the case, e.g., if `a` and `b` are unmodified copies of the same container), then the average-case complexity for `unordered_multiset` and `unordered_multimap` becomes proportional to N (but worst-case complexity remains $\mathcal{O}(N^2)$, e.g., for a pathologically bad hash function). The behavior of a program that uses `operator==` or `operator!=` on unordered containers is undefined unless the `Pred` function object has the same behavior for both containers and the equality comparison function for `Key` is a refinement[227] of the partition into equivalent-key groups produced by `Pred`.

13 The iterator types `iterator` and `const_iterator` of an unordered associative container are of at least the forward iterator category. For unordered associative containers where the key type and value type are the same, both `iterator` and `const_iterator` are constant iterators.

14 The `insert` and `emplace` members shall not affect the validity of references to container elements, but may invalidate all iterators to the container. The `erase` members shall invalidate only iterators and references to the erased elements, and preserve the relative order of the elements that are not erased.

15 The `insert` and `emplace` members shall not affect the validity of iterators if `(N+n) <= z * B`, where `N` is the number of elements in the container prior to the insert operation, `n` is the number of elements inserted, `B` is the container's bucket count, and `z` is the container's maximum load factor.

16 The `extract` members invalidate only iterators to the removed element, and preserve the relative order of the elements that are not erased; pointers and references to the removed element remain valid. However, accessing the element through such pointers and references while the element is owned by a `node_type` is undefined behavior. References and pointers to an element obtained while it is owned by a `node_type` are invalidated if the element is successfully inserted.

17 If the *qualified-id* `Hash::transparent_key_equal` is valid and denotes a type (**??**), then the program is ill-formed if either:

(17.1) — *qualified-id* `Hash::transparent_key_equal::is_transparent` is not valid or does not denote a type, or

(17.2) — `Pred` is a different type than `equal_to<Key>` or `Hash::transparent_key_equal`.

The member function templates `find`, `count`, `equal_range`, and `contains` shall not participate in overload resolution unless the *qualified-id* `Hash::transparent_key_equal` is valid and denotes a type (**??**).

18 A deduction guide for an unordered associative container shall not participate in overload resolution if any of the following are true:

(18.1) — It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.

(18.2) — It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.

227) Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.

(18.3) — It has a `Hash` template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.

(18.4) — It has a `Pred` template parameter and a type that qualifies as an allocator is deduced for that parameter.

21.2.7.1 Exception safety guarantees [unord.req.except]

1 For unordered associative containers, no `clear()` function throws an exception. `erase(k)` does not throw an exception unless that exception is thrown by the container's `Hash` or `Pred` object (if any).

2 For unordered associative containers, if an exception is thrown by any operation other than the container's hash function from within an `insert` or `emplace` function inserting a single element, the insertion has no effect.

3 For unordered associative containers, no `swap` function throws an exception unless that exception is thrown by the swap of the container's `Hash` or `Pred` object (if any).

4 For unordered associative containers, if an exception is thrown from within a `rehash()` function other than by the container's hash function or comparison function, the `rehash()` function has no effect.

21.3 Sequence containers [sequences]

21.3.1 In general [sequences.general]

1 The headers `<array>`, `<deque>`, `<forward_list>`, `<list>`, and `<vector>` define class templates that meet the requirements for sequence containers.

2 The following exposition-only alias template may appear in deduction guides for sequence containers:

```
template<class InputIterator>
  using iter-value-type = typename iterator_traits<InputIterator>::value_type;  // exposition only
```

21.3.2 Header <array> synopsis [array.syn]

```
#include <initializer_list>

namespace std {
  // 21.3.7, class template array
  template<class T, size_t N> struct array;

  template<class T, size_t N>
    constexpr bool operator==(const array<T, N>& x, const array<T, N>& y);
  template<class T, size_t N>
    constexpr bool operator!=(const array<T, N>& x, const array<T, N>& y);
  template<class T, size_t N>
    constexpr bool operator< (const array<T, N>& x, const array<T, N>& y);
  template<class T, size_t N>
    constexpr bool operator> (const array<T, N>& x, const array<T, N>& y);
  template<class T, size_t N>
    constexpr bool operator<=(const array<T, N>& x, const array<T, N>& y);
  template<class T, size_t N>
    constexpr bool operator>=(const array<T, N>& x, const array<T, N>& y);
  template<class T, size_t N>
    constexpr void swap(array<T, N>& x, array<T, N>& y) noexcept(noexcept(x.swap(y)));

  template<class T> class tuple_size;
  template<size_t I, class T> class tuple_element;
  template<class T, size_t N>
    struct tuple_size<array<T, N>>;
  template<size_t I, class T, size_t N>
    struct tuple_element<I, array<T, N>>;
  template<size_t I, class T, size_t N>
    constexpr T& get(array<T, N>&) noexcept;
  template<size_t I, class T, size_t N>
    constexpr T&& get(array<T, N>&&) noexcept;
  template<size_t I, class T, size_t N>
    constexpr const T& get(const array<T, N>&) noexcept;
```

```
  template<size_t I, class T, size_t N>
    constexpr const T&& get(const array<T, N>&&) noexcept;
}
```

21.3.3 Header <deque> synopsis [deque.syn]

```
#include <initializer_list>

namespace std {
  // 21.3.8, class template deque
  template<class T, class Allocator = allocator<T>> class deque;

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

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

  template <class T, class Allocator, class U>
    void erase(deque<T, Allocator>& c, const U& value);
  template <class T, class Allocator, class Predicate>
    void erase_if(deque<T, Allocator>& c, Predicate pred);

  namespace pmr {
    template<class T>
      using deque = std::deque<T, polymorphic_allocator<T>>;
  }
}
```

21.3.4 Header <forward_list> synopsis [forward__list.syn]

```
#include <initializer_list>

namespace std {
  // 21.3.9, class template forward_list
  template<class T, class Allocator = allocator<T>> class forward_list;

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

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

```
  template <class T, class Allocator, class U>
    void erase(forward_list<T, Allocator>& c, const U& value);
  template <class T, class Allocator, class Predicate>
    void erase_if(forward_list<T, Allocator>& c, Predicate pred);

  namespace pmr {
    template<class T>
      using forward_list = std::forward_list<T, polymorphic_allocator<T>>;
  }
}
```

21.3.5 Header <list> synopsis [list.syn]

```
#include <initializer_list>

namespace std {
  // 21.3.10, class template list
  template<class T, class Allocator = allocator<T>> class list;

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

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

  template <class T, class Allocator, class U>
    void erase(list<T, Allocator>& c, const U& value);
  template <class T, class Allocator, class Predicate>
    void erase_if(list<T, Allocator>& c, Predicate pred);

  namespace pmr {
    template<class T>
      using list = std::list<T, polymorphic_allocator<T>>;
  }
}
```

21.3.6 Header <vector> synopsis [vector.syn]

```
#include <initializer_list>

namespace std {
  // 21.3.11, class template vector
  template<class T, class Allocator = allocator<T>> class vector;

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

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

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

  template <class T, class Allocator, class U>
    void erase(vector<T, Allocator>& c, const U& value);
  template <class T, class Allocator, class Predicate>
    void erase_if(vector<T, Allocator>& c, Predicate pred);

  // 21.3.12, class vector<bool>
  template<class Allocator> class vector<bool, Allocator>;

  // hash support
  template<class T> struct hash;
  template<class Allocator> struct hash<vector<bool, Allocator>>;

  namespace pmr {
    template<class T>
      using vector = std::vector<T, polymorphic_allocator<T>>;
  }
}
```

21.3.7 Class template array [array]

21.3.7.1 Overview [array.overview]

1 The header `<array>` defines a class template for storing fixed-size sequences of objects. An `array` is a contiguous container (21.2.1). An instance of `array<T, N>` stores `N` elements of type `T`, so that `size() == N` is an invariant.

2 An `array` is an aggregate (**??**) that can be list-initialized with up to `N` elements whose types are convertible to `T`.

3 An `array` satisfies all of the requirements of a container and of a reversible container (21.2), except that a default constructed `array` object is not empty and that `swap` does not have constant complexity. An `array` satisfies some of the requirements of a sequence container (21.2.3). Descriptions are provided here only for operations on `array` that are not described in one of these tables and for operations where there is additional semantic information.

4 The types `iterator` and `const_iterator` satisfy the constexpr iterator requirements (**??**).

```
namespace std {
  template<class T, size_t N>
  struct array {
    // types
    using value_type             = T;
    using pointer                = T*;
    using const_pointer          = const T*;
    using reference              = T&;
    using const_reference        = const T&;
    using size_type              = size_t;
    using difference_type        = ptrdiff_t;
    using iterator               = implementation-defined; // see 21.2
    using const_iterator         = implementation-defined; // see 21.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

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

    constexpr void fill(const T& u);
    constexpr void swap(array&) noexcept(is_nothrow_swappable_v<T>);
```

```
    // iterators
    constexpr iterator               begin() noexcept;
    constexpr const_iterator         begin() const noexcept;
    constexpr iterator               end() noexcept;
    constexpr const_iterator         end() const noexcept;

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

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

    // capacity
    [[nodiscard]] constexpr bool empty() const noexcept;
    constexpr size_type size() const noexcept;
    constexpr size_type max_size() const noexcept;

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

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

  template<class T, class... U>
    array(T, U...) -> array<T, 1 + sizeof...(U)>;
}
```

21.3.7.2 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 21.2. In addition to the requirements specified in the container requirements table, the implicit move constructor and move assignment operator for `array` require that `T` be *Cpp17MoveConstructible* or *Cpp17MoveAssignable*, respectively.

```
template<class T, class... U>
  array(T, U...) -> array<T, 1 + sizeof...(U)>;
```

2 ~~*Requires:*~~ *Mandates:* `(is_same_v<T, U> && ...)` ~~is~~shall be `true`. ~~Otherwise the program is ill-formed.~~

21.3.7.3 Member functions [array.members]

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

1 *Returns:* `N`.

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

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

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

3 *Effects:* As if by `fill_n(begin(), N, u)`.

```
constexpr void swap(array& y) noexcept(is_nothrow_swappable_v<T>);
```

4 *Effects:* Equivalent to `swap_ranges(begin(), end(), y.begin())`.

5 [*Note*: Unlike the `swap` function for other containers, `array::swap` takes linear time, may exit via an exception, and does not cause iterators to become associated with the other container. — *end note*]

21.3.7.4 Specialized algorithms [array.special]

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

1 ~~*Remarks:* This function shall not participate in overload resolution unless `N == 0` or `is_swappable_v<T>` is `true`.~~

2 *Constraints:* `N == 0` or `is_swappable_v<T>` is `true`.

3 *Effects:* As if by `x.swap(y)`.

4 *Complexity:* Linear in `N`.

21.3.7.5 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 undefined.

4 Member function `swap()` shall have a non-throwing exception specification.

21.3.7.6 Tuple interface [array.tuple]

```
template<class T, size_t N>
  struct tuple_size<array<T, N>> : integral_constant<size_t, N> { };

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

1 *Mandates:* `I < N`.

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

3 *Value:* The type T.

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

4 *Mandates:* `I < N`.

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

6 *Returns:* A reference to the $\texttt{I}^{\text{th}}$ element of `a`, where indexing is zero-based.

21.3.8 Class template deque [deque]

21.3.8.1 Overview [deque.overview]

1 A `deque` is a sequence container that 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. Storage management is handled automatically.

2 A `deque` satisfies all of the requirements of a container, of a reversible container (given in tables in 21.2), of a sequence container, including the optional sequence container requirements (21.2.3), and of an allocator-aware container (Table 65). 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 T, class Allocator = allocator<T>>
  class deque {
  public:
    // types
    using value_type             = T;
    using allocator_type         = Allocator;
    using pointer                = typename allocator_traits<Allocator>::pointer;
    using const_pointer          = typename allocator_traits<Allocator>::const_pointer;
    using reference              = value_type&;
    using const_reference        = const value_type&;
    using size_type              = implementation-defined; // see 21.2
    using difference_type        = implementation-defined; // see 21.2
    using iterator               = implementation-defined; // see 21.2
    using const_iterator         = implementation-defined; // see 21.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    // 21.3.8.2, construct/copy/destroy
    deque() : deque(Allocator()) { }
    explicit deque(const Allocator&);
    explicit deque(size_type n, const Allocator& = Allocator());
    deque(size_type n, const T& value, const Allocator& = Allocator());
    template<class InputIterator>
      deque(InputIterator first, InputIterator last, const Allocator& = Allocator());
    deque(const deque& x);
    deque(deque&&);
    deque(const deque&, const Allocator&);
    deque(deque&&, const Allocator&);
    deque(initializer_list<T>, const Allocator& = Allocator());

    ~deque();
    deque& operator=(const deque& x);
    deque& operator=(deque&& x)
      noexcept(allocator_traits<Allocator>::is_always_equal::value);
    deque& operator=(initializer_list<T>);
    template<class InputIterator>
      void assign(InputIterator first, InputIterator last);
    void assign(size_type n, const T& t);
    void assign(initializer_list<T>);
    allocator_type get_allocator() const noexcept;

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

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

    // 21.3.8.3, capacity
    [[nodiscard]] bool empty() const noexcept;
    size_type size() const noexcept;
    size_type max_size() const noexcept;
    void      resize(size_type sz);
    void      resize(size_type sz, const T& c);
    void      shrink_to_fit();
```

```
    // 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;

    // 21.3.8.4, modifiers
    template<class... Args> reference emplace_front(Args&&... args);
    template<class... Args> reference emplace_back(Args&&... args);
    template<class... Args> iterator emplace(const_iterator position, Args&&... args);

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

    iterator insert(const_iterator position, const T& x);
    iterator insert(const_iterator position, T&& x);
    iterator insert(const_iterator position, size_type n, const T& x);
    template<class InputIterator>
      iterator insert(const_iterator position, InputIterator first, InputIterator last);
    iterator insert(const_iterator position, initializer_list<T>);

    void pop_front();
    void pop_back();

    iterator erase(const_iterator position);
    iterator erase(const_iterator first, const_iterator last);
    void     swap(deque&)
      noexcept(allocator_traits<Allocator>::is_always_equal::value);
    void     clear() noexcept;
  };

  template<class InputIterator, class Allocator = allocator<iter-value-type<InputIterator>>>
    deque(InputIterator, InputIterator, Allocator = Allocator())
      -> deque<iter-value-type<InputIterator>, Allocator>;

  // swap
  template<class T, class Allocator>
    void swap(deque<T, Allocator>& x, deque<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));
}
```

21.3.8.2 Constructors, copy, and assignment [deque.cons]

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

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

2 *Complexity:* Constant.

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

3 *Effects:* Constructs a `deque` with `n` default-inserted elements using the specified allocator.

4 ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17DefaultInsertable* into `*this`.

5 *Complexity:* Linear in `n`.

```
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:*~~ *Expects:* `T` shall be *Cpp17CopyInsertable* into `*this`.

8 *Complexity:* Linear in `n`.

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

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

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

21.3.8.3 Capacity [deque.capacity]

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

1 *Effects:* If `sz < size()`, erases the last `size() - sz` elements from the sequence. Otherwise, appends `sz - size()` default-inserted elements to the sequence.

2 ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17MoveInsertable* and *Cpp17DefaultInsertable* into `*this`.

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

3 *Effects:* If `sz < size()`, erases the last `size() - sz` elements from the sequence. Otherwise, appends `sz - size()` copies of `c` to the sequence.

4 ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17CopyInsertable* into `*this`.

```
void shrink_to_fit();
```

5 ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17MoveInsertable* into `*this`.

6 *Effects:* `shrink_to_fit` is a non-binding request to reduce memory use but does not change the size of the sequence. [*Note*: The request is non-binding to allow latitude for implementation-specific optimizations. — *end note*] If an exception is thrown other than by the move constructor of a non-*Cpp17CopyInsertable* `T` there are no effects.

7 *Complexity:* Linear in the size of the sequence.

8 *Remarks:* `shrink_to_fit` invalidates all the references, pointers, and iterators referring to the elements in the sequence as well as the past-the-end iterator.

21.3.8.4 Modifiers [deque.modifiers]

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template<class InputIterator>
  iterator insert(const_iterator position,
                  InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);

template<class... Args> reference emplace_front(Args&&... args);
template<class... Args> reference emplace_back(Args&&... args);
template<class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
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, move constructor, assignment operator, or move assignment operator of `T` there are no effects. If an exception is thrown while inserting a single element at either end, there are no effects. Otherwise, if an exception is thrown by the move constructor of a non-*Cpp17CopyInsertable* `T`, the effects are unspecified.

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`.

```
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void pop_front();
void pop_back();
```

4 *Effects:* An erase operation that erases the last element of a deque invalidates only the past-the-end iterator and all iterators and references to the erased elements. An erase operation that erases the first element of a deque but not the last element invalidates only iterators and references to the erased elements. An erase operation that erases neither the first element nor the last element of a deque invalidates the past-the-end iterator and all iterators and references to all the elements of the deque. [*Note*: `pop_front` and `pop_back` are erase operations. — *end note*]

5 *Complexity:* The number of calls to the destructor of `T` is the same as the number of elements erased, but the number of calls to the assignment operator of `T` is no more than the lesser 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 assignment operator of `T`.

21.3.8.5 Erasure [deque.erasure]

```
template <class T, class Allocator, class U>
  void erase(deque<T, Allocator>& c, const U& value);
```

1 *Effects:* Equivalent to: `c.erase(remove(c.begin(), c.end(), value), c.end());`

```
template <class T, class Allocator, class Predicate>
  void erase_if(deque<T, Allocator>& c, Predicate pred);
```

2 *Effects:* Equivalent to: `c.erase(remove_if(c.begin(), c.end(), pred), c.end());`

21.3.9 Class template `forward_list` [forwardlist]

21.3.9.1 Overview [forwardlist.overview]

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 62), except that the `size()` member function is not provided and `operator==` has linear complexity. A `forward_list` also satisfies all of the requirements for an allocator-aware container (Table 65). In addition, a `forward_list` provides the `assign` member functions (Table 66) and several of the optional container requirements (Table 67). 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.

3 [*Note*: Modifying any list requires access to the element preceding the first element of interest, but in a `forward_list` there is no constant-time way to access a preceding element. For this reason, ranges that are modified, such as those supplied to `erase` and `splice`, must be open at the beginning. — *end note*]

```
namespace std {
  template<class T, class Allocator = allocator<T>>
  class forward_list {
  public:
    // types
    using value_type      = T;
    using allocator_type  = Allocator;
    using pointer         = typename allocator_traits<Allocator>::pointer;
    using const_pointer   = typename allocator_traits<Allocator>::const_pointer;
    using reference       = value_type&;
    using const_reference = const value_type&;
    using size_type       = implementation-defined; // see 21.2
    using difference_type = implementation-defined; // see 21.2
    using iterator        = implementation-defined; // see 21.2
    using const_iterator  = implementation-defined; // see 21.2
```

```
// 21.3.9.2, construct/copy/destroy
forward_list() : forward_list(Allocator()) { }
explicit forward_list(const Allocator&);
explicit forward_list(size_type n, const Allocator& = Allocator());
forward_list(size_type n, const T& value, const Allocator& = Allocator());
template<class InputIterator>
  forward_list(InputIterator first, InputIterator last, const Allocator& = Allocator());
forward_list(const forward_list& x);
forward_list(forward_list&& x);
forward_list(const forward_list& x, const Allocator&);
forward_list(forward_list&& x, const Allocator&);
forward_list(initializer_list<T>, const Allocator& = Allocator());
~forward_list();
forward_list& operator=(const forward_list& x);
forward_list& operator=(forward_list&& x)
  noexcept(allocator_traits<Allocator>::is_always_equal::value);
forward_list& operator=(initializer_list<T>);
template<class InputIterator>
  void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
void assign(initializer_list<T>);
allocator_type get_allocator() const noexcept;

// 21.3.9.3, iterators
iterator before_begin() noexcept;
const_iterator before_begin() const noexcept;
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;

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

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type max_size() const noexcept;

// 21.3.9.4, element access
reference front();
const_reference front() const;

// 21.3.9.5, modifiers
template<class... Args> reference emplace_front(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void pop_front();

template<class... Args> iterator emplace_after(const_iterator position, Args&&... args);
iterator insert_after(const_iterator position, const T& x);
iterator insert_after(const_iterator position, T&& x);

iterator insert_after(const_iterator position, size_type n, const T& x);
template<class InputIterator>
  iterator insert_after(const_iterator position, InputIterator first, InputIterator last);
iterator insert_after(const_iterator position, initializer_list<T> il);

iterator erase_after(const_iterator position);
iterator erase_after(const_iterator position, const_iterator last);
void swap(forward_list&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value);
```

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

    // 21.3.9.6, forward_list operations
    void splice_after(const_iterator position, forward_list& x);
    void splice_after(const_iterator position, forward_list&& x);
    void splice_after(const_iterator position, forward_list& x, const_iterator i);
    void splice_after(const_iterator position, forward_list&& x, const_iterator i);
    void splice_after(const_iterator position, forward_list& x,
                      const_iterator first, const_iterator last);
    void splice_after(const_iterator position, forward_list&& x,
                      const_iterator first, const_iterator last);

    size_type remove(const T& value);
    template<class Predicate> size_type remove_if(Predicate pred);

    size_type unique();
    template<class BinaryPredicate> size_type unique(BinaryPredicate binary_pred);

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

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

    void reverse() noexcept;
  };

  template<class InputIterator, class Allocator = allocator<iter-value-type<InputIterator>>>
    forward_list(InputIterator, InputIterator, Allocator = Allocator())
      -> forward_list<iter-value-type<InputIterator>, Allocator>;

  // swap
  template<class T, class Allocator>
    void swap(forward_list<T, Allocator>& x, forward_list<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));
}
```

4 An incomplete type `T` may be used when instantiating `forward_list` if the allocator satisfies the allocator completeness requirements (**??**). `T` shall be complete before any member of the resulting specialization of `forward_list` is referenced.

21.3.9.2 Constructors, copy, and assignment [forwardlist.cons]

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

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

2 *Complexity:* Constant.

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

3 *Effects:* Constructs a `forward_list` object with `n` default-inserted elements using the specified allocator.

4 ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17DefaultInsertable* into `*this`.

5 *Complexity:* Linear in `n`.

```
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:*~~ *Expects:* `T` shall be *Cpp17CopyInsertable* into `*this`.

8 *Complexity:* Linear in `n`.

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

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

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

21.3.9.3 Iterators [forwardlist.iter]

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

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

2 *Effects:* `cbefore_begin()` is equivalent to `const_cast<forward_list const&>(*this).before_begin()`.

3 *Remarks:* `before_begin() == end()` shall equal `false`.

21.3.9.4 Element access [forwardlist.access]

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

1 *Returns:* `*begin()`

21.3.9.5 Modifiers [forwardlist.modifiers]

1 None of the overloads of `insert_after` shall affect the validity of iterators and references, and `erase_after` shall invalidate only iterators and references to the erased elements. If an exception is thrown during `insert_after` there shall be no effect. Inserting `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 in `n` and the number of calls to the destructor of type `T` is exactly equal to `n`.

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

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

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

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

```
void pop_front();
```

4 *Effects:* As if by `erase_after(before_begin())`.

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

5 ~~*Requires:*~~ *Expects:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`.

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

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

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

8 ~~*Requires:*~~ *Expects:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`.

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

10 *Returns:* An iterator pointing to the last inserted copy of `x` or `position` if `n == 0`.

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

11 ~~*Requires:*~~ *Expects:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`. `first` and `last` are not iterators in `*this`.

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

13 *Returns:* An iterator pointing to the last inserted element or `position` if `first == last`.

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

14 *Effects:* `insert_after(p, il.begin(), il.end())`.

15 *Returns:* An iterator pointing to the last inserted element or `position` if `il` is empty.

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

16 ~~*Requires:*~~ *Expects:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`.

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

18 *Returns:* An iterator pointing to the new object.

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

19 ~~*Requires:*~~ *Expects:* The iterator following `position` is dereferenceable.

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

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

22 *Throws:* Nothing.

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

23 ~~*Requires:*~~ *Expects:* All iterators in the range `(position, last)` are dereferenceable.

24 *Effects:* Erases the elements in the range `(position, last)`.

25 *Returns:* `last`.

26 *Throws:* Nothing.

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

27 *Expects:* `T` shall be *Cpp17DefaultInsertable* into `*this`.

28 *Effects:* If `sz < distance(begin(), end())`, erases the last `distance(begin(), end()) - sz` elements from the list. Otherwise, inserts `sz - distance(begin(), end())` default-inserted elements at the end of the list.

29 ~~*Requires:* `T` shall be *Cpp17DefaultInsertable* into `*this`.~~

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

30 *Expects:* `T` shall be *Cpp17CopyInsertable* into `*this`.

31 *Effects:* If `sz < distance(begin(), end())`, erases the last `distance(begin(), end()) - sz` elements from the list. Otherwise, inserts `sz - distance(begin(), end())` copies of `c` at the end of the list.

32 ~~*Requires:* `T` shall be *Cpp17CopyInsertable* into `*this`.~~

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

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

34 *Remarks:* Does not invalidate past-the-end iterators.

21.3.9.6 Operations [forwardlist.ops]

1 In this subclause, arguments for a template parameter named `Predicate` or `BinaryPredicate` shall satisfy the corresponding requirements in **??**. For `merge` and `sort`, the definitions and requirements in **??** apply.

```
void splice_after(const_iterator position, forward_list& x);
```

```
void splice_after(const_iterator position, forward_list&& x);
```

2 ~~*Requires:*~~ *Expects:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`. `get_allocator() == x.get_allocator()`. `addressof(x) != this`.

3 *Effects:* Inserts the contents of `x` after `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`.

4 *Throws:* Nothing.

5 *Complexity:* $\mathcal{O}$(`distance(x.begin(), x.end())`)

```
void splice_after(const_iterator position, forward_list& x, const_iterator i);
void splice_after(const_iterator position, forward_list&& x, const_iterator i);
```

6 ~~*Requires:*~~ *Expects:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`. The iterator following `i` is a dereferenceable iterator in `x`. `get_allocator() == x.get_allocator()`.

7 *Effects:* Inserts the element following `i` into `*this`, following `position`, and removes it from `x`. The result is unchanged if `position == i` or `position == ++i`. Pointers and references to `*++i` continue to refer to the same element but as a member of `*this`. Iterators to `*++i` continue to refer to the same element, but now behave as iterators into `*this`, not into `x`.

8 *Throws:* Nothing.

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

```
void splice_after(const_iterator position, forward_list& x,
                  const_iterator first, const_iterator last);
void splice_after(const_iterator position, forward_list&& x,
                  const_iterator first, const_iterator last);
```

10 ~~*Requires:*~~ *Expects:* `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`. `(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)`. `get_allocator() == x.get_allocator()`.

11 *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`.

12 *Complexity:* $\mathcal{O}$(`distance(first, last)`)

```
size_type remove(const T& value);
template<class Predicate> size_type remove_if(Predicate pred);
```

13 *Effects:* Erases all the elements in the list referred to by a list iterator `i` for which the following conditions hold: `*i == value` (for `remove()`), `pred(*i)` is `true` (for `remove_if()`). Invalidates only the iterators and references to the erased elements.

14 *Returns:* The number of elements erased.

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

16 *Remarks:* Stable (**??**).

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

```
size_type unique();
template<class BinaryPredicate> size_type unique(BinaryPredicate pred);
```

18 *Effects:* Erases 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. Invalidates only the iterators and references to the erased elements.

19 *Returns:* The number of elements erased.

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

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.

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

22 ~~*Requires:*~~ *Expects:* `*this` and `x` are both sorted with respect to the comparator `operator<` (for the first two overloads) or `comp` (for the last two overloads), and `get_allocator() == x.get_allocator()` is `true`.

23 *Effects:* Merges the two sorted ranges `[begin(), end())` and `[x.begin(), x.end())`. `x` is empty after the merge. If an exception is thrown other than by a comparison there are no effects. 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`.

24 *Remarks:* Stable (**??**). The behavior is undefined if `get_allocator() != x.get_allocator()`.

25 *Complexity:* At most `distance(begin(), end()) + distance(x.begin(), x.end()) - 1` comparisons.

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

26 *Effects:* Sorts the list according to the `operator<` or the `comp` function object. If an exception is thrown, the order of the elements in `*this` is unspecified. Does not affect the validity of iterators and references.

27 *Remarks:* Stable (**??**).

28 *Complexity:* Approximately $N \log N$ comparisons, where N is `distance(begin(), end())`.

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

29 *Effects:* Reverses the order of the elements in the list. Does not affect the validity of iterators and references.

30 *Complexity:* Linear time.

21.3.9.7 Erasure [forward_list.erasure]

```
template <class T, class Allocator, class U>
  void erase(forward_list<T, Allocator>& c, const U& value);
```

1 *Effects:* Equivalent to: `erase_if(c, [&](auto& elem)  return elem == value; );`

```
template <class T, class Allocator, class Predicate>
  void erase_if(forward_list<T, Allocator>& c, Predicate pred);
```

2 *Effects:* Equivalent to: `c.remove_if(pred);`

21.3.10 Class template `list` [list]

21.3.10.1 Overview [list.overview]

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 (21.3.11) and deques (21.3.8), 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 21.2), of a sequence container, including most of the optional sequence container requirements (21.2.3), and of an allocator-aware container (Table 65). The exceptions are the `operator[]` and `at` member functions, which are not provided.[228] 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.

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

```
namespace std {
  template<class T, class Allocator = allocator<T>>
  class list {
  public:
    // types
    using value_type             = T;
    using allocator_type         = Allocator;
    using pointer                = typename allocator_traits<Allocator>::pointer;
    using const_pointer          = typename allocator_traits<Allocator>::const_pointer;
    using reference              = value_type&;
    using const_reference        = const value_type&;
    using size_type              = implementation-defined; // see 21.2
    using difference_type        = implementation-defined; // see 21.2
    using iterator               = implementation-defined; // see 21.2
    using const_iterator         = implementation-defined; // see 21.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    // 21.3.10.2, construct/copy/destroy
    list() : list(Allocator()) { }
    explicit list(const Allocator&);
    explicit list(size_type n, const Allocator& = Allocator());
    list(size_type n, const T& value, const Allocator& = Allocator());
    template<class InputIterator>
      list(InputIterator first, InputIterator last, const Allocator& = Allocator());
    list(const list& x);
    list(list&& x);
    list(const list&, const Allocator&);
    list(list&&, const Allocator&);
    list(initializer_list<T>, const Allocator& = Allocator());
    ~list();
    list& operator=(const list& x);
    list& operator=(list&& x)
      noexcept(allocator_traits<Allocator>::is_always_equal::value);
    list& operator=(initializer_list<T>);
    template<class InputIterator>
      void assign(InputIterator first, InputIterator last);
    void assign(size_type n, const T& t);
    void assign(initializer_list<T>);
    allocator_type get_allocator() const noexcept;

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

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

    // 21.3.10.3, capacity
    [[nodiscard]] bool empty() const noexcept;
    size_type size() const noexcept;
    size_type max_size() const noexcept;
    void      resize(size_type sz);
    void      resize(size_type sz, const T& c);
```

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

    // 21.3.10.4, modifiers
    template<class... Args> reference emplace_front(Args&&... args);
    template<class... Args> reference emplace_back(Args&&... args);
    void push_front(const T& x);
    void push_front(T&& x);
    void pop_front();
    void push_back(const T& x);
    void push_back(T&& x);
    void pop_back();

    template<class... Args> iterator emplace(const_iterator position, Args&&... args);
    iterator insert(const_iterator position, const T& x);
    iterator insert(const_iterator position, T&& x);
    iterator insert(const_iterator position, size_type n, const T& x);
    template<class InputIterator>
      iterator insert(const_iterator position, InputIterator first, InputIterator last);
    iterator insert(const_iterator position, initializer_list<T> il);

    iterator erase(const_iterator position);
    iterator erase(const_iterator position, const_iterator last);
    void     swap(list&) noexcept(allocator_traits<Allocator>::is_always_equal::value);
    void     clear() noexcept;

    // 21.3.10.5, list operations
    void splice(const_iterator position, list& x);
    void splice(const_iterator position, list&& x);
    void splice(const_iterator position, list& x, const_iterator i);
    void splice(const_iterator position, list&& x, const_iterator i);
    void splice(const_iterator position, list& x, const_iterator first, const_iterator last);
    void splice(const_iterator position, list&& x, const_iterator first, const_iterator last);

    size_type remove(const T& value);
    template<class Predicate> size_type remove_if(Predicate pred);

    size_type unique();
    template<class BinaryPredicate>
      size_type unique(BinaryPredicate binary_pred);

    void merge(list& x);
    void merge(list&& x);
    template<class Compare> void merge(list& x, Compare comp);
    template<class Compare> void merge(list&& x, Compare comp);

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

    void reverse() noexcept;
  };

  template<class InputIterator, class Allocator = allocator<iter-value-type<InputIterator>>>
    list(InputIterator, InputIterator, Allocator = Allocator())
      -> list<iter-value-type<InputIterator>, Allocator>;

  // swap
  template<class T, class Allocator>
    void swap(list<T, Allocator>& x, list<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));
}
```

3 An incomplete type `T` may be used when instantiating `list` if the allocator satisfies the allocator completeness requirements (**??**). `T` shall be complete before any member of the resulting specialization of `list` is referenced.

21.3.10.2 Constructors, copy, and assignment [list.cons]

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

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

2 *Complexity:* Constant.

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

3 *Expects:* `T` shall be *Cpp17DefaultInsertable* into `*this`.

4 *Effects:* Constructs a `list` with `n` default-inserted elements using the specified allocator.

5 ~~*Requires:* `T` shall be *Cpp17DefaultInsertable* into `*this`.~~

6 *Complexity:* Linear in `n`.

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

7 *Expects:* `T` shall be *Cpp17CopyInsertable* into `*this`.

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

9 ~~*Requires:* `T` shall be *Cpp17CopyInsertable* into `*this`.~~

10 *Complexity:* Linear in `n`.

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

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

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

21.3.10.3 Capacity [list.capacity]

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

1 *Expects:* `T` shall be *Cpp17DefaultInsertable* into `*this`.

2 *Effects:* If `size() < sz`, appends `sz - size()` default-inserted elements to the sequence. If `sz <= size()`, equivalent to:

```
list<T>::iterator it = begin();
advance(it, sz);
erase(it, end());
```

3 ~~*Requires:* `T` shall be *Cpp17DefaultInsertable* into `*this`.~~

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

4 *Expects:* `T` shall be *Cpp17CopyInsertable* into `*this`.

5 *Effects:* As if by:

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

6 ~~*Requires:* `T` shall be *Cpp17CopyInsertable* into `*this`.~~

21.3.10.4 Modifiers [list.modifiers]

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

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

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

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.

21.3.10.5 Operations [list.ops]

1 Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for them.[229] In this subclause, arguments for a template parameter named `Predicate` or `BinaryPredicate` shall satisfy the corresponding requirements in **??**. For `merge` and `sort`, the definitions and requirements in **??** apply.

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& x);
void splice(const_iterator position, list&& x);
```

3 ~~*Requires:*~~ *Expects:* `addressof(x) != 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& x, const_iterator i);
void splice(const_iterator position, list&& x, const_iterator i);
```

7 ~~*Requires:*~~ *Expects:* `i` is a valid dereferenceable iterator of `x`.

8 *Effects:* Inserts an element pointed to by `i` from list `x` before `position` and removes the element from `x`. The result is unchanged if `position == i` or `position == ++i`. Pointers and references to `*i`

229) As specified in **??**, the requirements in this Clause apply only to lists whose allocators compare equal.

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`.

9 *Throws:* Nothing.

10 *Complexity:* Constant time.

```
void splice(const_iterator position, list& x, const_iterator first,
            const_iterator last);
void splice(const_iterator position, list&& x, const_iterator first,
            const_iterator last);
```

11 ~~*Requires:*~~ *Expects:* `[first, last)` is a valid range in `x`. ~~The program has undefined behavior if~~ `position` is not an iterator in the range `[first, last)`.

12 *Effects:* Inserts elements in the range `[first, last)` before `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`.

13 *Throws:* Nothing.

14 *Complexity:* Constant time if `&x == this`; otherwise, linear time.

```
size_type remove(const T& value);
template<class Predicate> size_type remove_if(Predicate pred);
```

15 *Effects:* Erases all the elements in the list referred to by a list iterator `i` for which the following conditions hold: `*i == value`, `pred(*i) != false`. Invalidates only the iterators and references to the erased elements.

16 *Returns:* The number of elements erased.

17 *Throws:* Nothing unless an exception is thrown by `*i == value` or `pred(*i) != false`.

18 *Remarks:* Stable (**??**).

19 *Complexity:* Exactly `size()` applications of the corresponding predicate.

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

20 *Effects:* Erases 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 of `unique` with no arguments) or `pred(*i, *(i - 1))` (for the version of `unique` with a predicate argument) holds. Invalidates only the iterators and references to the erased elements.

21 *Returns:* The number of elements erased.

22 *Throws:* Nothing unless an exception is thrown by `*i == *(i-1)` or `pred(*i, *(i - 1))`

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

```
void merge(list& x);
void merge(list&& x);
template<class Compare> void merge(list& x, Compare comp);
template<class Compare> void merge(list&& x, Compare comp);
```

24 ~~*Requires:*~~ *Expects:* Both the list and the argument list shall be sorted with respect to the comparator `operator<` (for the first two overloads) or `comp` (for the last two overloads). `get_allocator() == x.get_allocator()`.

25 *Effects:* If `addressof(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`. 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`.

26 *Remarks:* Stable (**??**). If `addressof(x) != this`, the range `[x.begin(), x.end())` is empty after the merge. No elements are copied by this operation. ~~The behavior is undefined if `get_allocator() != x.get_allocator`~~

27 *Complexity:* At most `size() + x.size() - 1` applications of `comp` if `addressof(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() noexcept;
```

28 *Effects:* Reverses the order of the elements in the list. Does not affect the validity of iterators and references.

29 *Complexity:* Linear time.

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

30 *Effects:* Sorts the list according to the `operator<` or a `Compare` function object. If an exception is thrown, the order of the elements in `*this` is unspecified. Does not affect the validity of iterators and references.

31 *Remarks:* Stable (**??**).

32 *Complexity:* Approximately $N \log N$ comparisons, where `N == size()`.

21.3.10.6 Erasure [list.erasure]

```
template <class T, class Allocator, class U>
  void erase(list<T, Allocator>& c, const U& value);
```

1 *Effects:* Equivalent to: `erase_if(c, [&](auto& elem)  return elem == value; );`

```
template <class T, class Allocator, class Predicate>
  void erase_if(list<T, Allocator>& c, Predicate pred);
```

2 *Effects:* Equivalent to: `c.remove_if(pred);`

21.3.11 Class template `vector` [vector]

21.3.11.1 Overview [vector.overview]

1 A `vector` is a sequence container that 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.

2 A `vector` satisfies all of the requirements of a container and of a reversible container (given in two tables in 21.2), of a sequence container, including most of the optional sequence container requirements (21.2.3), of an allocator-aware container (Table 65), and, for an element type other than `bool`, of a contiguous container (21.2.1). The exceptions are the `push_front`, `pop_front`, and `emplace_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 T, class Allocator = allocator<T>>
  class vector {
  public:
    // types
    using value_type             = T;
    using allocator_type         = Allocator;
    using pointer                = typename allocator_traits<Allocator>::pointer;
    using const_pointer          = typename allocator_traits<Allocator>::const_pointer;
    using reference              = value_type&;
    using const_reference        = const value_type&;
    using size_type              = implementation-defined; // see 21.2
    using difference_type        = implementation-defined; // see 21.2
    using iterator               = implementation-defined; // see 21.2
    using const_iterator         = implementation-defined; // see 21.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    // 21.3.11.2, construct/copy/destroy
    vector() noexcept(noexcept(Allocator())) : vector(Allocator()) { }
    explicit vector(const Allocator&) noexcept;
```

```
explicit vector(size_type n, const Allocator& = Allocator());
vector(size_type n, const T& value, const Allocator& = Allocator());
template<class InputIterator>
  vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
vector(const vector& x);
vector(vector&&) noexcept;
vector(const vector&, const Allocator&);
vector(vector&&, const Allocator&);
vector(initializer_list<T>, const Allocator& = Allocator());
~vector();
vector& operator=(const vector& x);
vector& operator=(vector&& x)
  noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
           allocator_traits<Allocator>::is_always_equal::value);
vector& operator=(initializer_list<T>);
template<class InputIterator>
  void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& u);
void assign(initializer_list<T>);
allocator_type get_allocator() const noexcept;

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

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

// 21.3.11.3, capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
size_type capacity() const noexcept;
void      resize(size_type sz);
void      resize(size_type sz, const T& c);
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;

// 21.3.11.4, data access
T*       data() noexcept;
const T* data() const noexcept;

// 21.3.11.5, modifiers
template<class... Args> reference emplace_back(Args&&... args);
void push_back(const T& x);
void push_back(T&& x);
```

```
    void pop_back();

    template<class... Args> iterator emplace(const_iterator position, Args&&... args);
    iterator insert(const_iterator position, const T& x);
    iterator insert(const_iterator position, T&& x);
    iterator insert(const_iterator position, size_type n, const T& x);
    template<class InputIterator>
      iterator insert(const_iterator position, InputIterator first, InputIterator last);
    iterator insert(const_iterator position, initializer_list<T> il);
    iterator erase(const_iterator position);
    iterator erase(const_iterator first, const_iterator last);
    void     swap(vector&)
      noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
               allocator_traits<Allocator>::is_always_equal::value);
    void     clear() noexcept;
  };

  template<class InputIterator, class Allocator = allocator<iter-value-type<InputIterator>>>
    vector(InputIterator, InputIterator, Allocator = Allocator())
      -> vector<iter-value-type<InputIterator>, Allocator>;

  // swap
  template<class T, class Allocator>
    void swap(vector<T, Allocator>& x, vector<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));
}
```

3 An incomplete type `T` may be used when instantiating `vector` if the allocator satisfies the allocator completeness requirements (**??**). `T` shall be complete before any member of the resulting specialization of `vector` is referenced.

21.3.11.2 Constructors, copy, and assignment [vector.cons]

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

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

2 *Complexity:* Constant.

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

3 *Expects:* `T` shall be *Cpp17DefaultInsertable* into `*this`.

4 *Effects:* Constructs a `vector` with `n` default-inserted elements using the specified allocator.

5 ~~*Requires:* `T` shall be *Cpp17DefaultInsertable* into `*this`.~~

6 *Complexity:* Linear in `n`.

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

7 *Expects:* `T` shall be *Cpp17CopyInsertable* into `*this`.

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

9 ~~*Requires:* `T` shall be *Cpp17CopyInsertable* into `*this`.~~

10 *Complexity:* Linear in `n`.

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

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

12 *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. It makes order `N` calls to the copy constructor of `T` and order $\log N$ reallocations if they are just input iterators.

21.3.11.3 Capacity [vector.capacity]

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

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

2 *Complexity:* Constant time.

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

3 ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17MoveInsertable* into `*this`.

4 *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 other than by the move constructor of a non-*Cpp17CopyInsertable* type, there are no effects.

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

6 *Throws:* `length_error` if `n > max_size()`.[230]

7 *Remarks:* Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. No reallocation shall take 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();
```

8 ~~*Requires:*~~ *Expects:* `T` shall be *Cpp17MoveInsertable* into `*this`.

9 *Effects:* `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*] It does not increase `capacity()`, but may reduce `capacity()` by causing reallocation. If an exception is thrown other than by the move constructor of a non-*Cpp17CopyInsertable* `T` there are no effects.

10 *Complexity:* Linear in the size of the sequence.

11 *Remarks:* Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence as well as the past-the-end iterator. If no reallocation happens, they remain valid.

```
void swap(vector& x)
  noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
           allocator_traits<Allocator>::is_always_equal::value);
```

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

13 *Complexity:* Constant time.

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

14 *Expects:* `T` shall be *Cpp17MoveInsertable* and *Cpp17DefaultInsertable* into `*this`.

15 *Effects:* If `sz < size()`, erases the last `size() - sz` elements from the sequence. Otherwise, appends `sz - size()` default-inserted elements to the sequence.

16 ~~*Requires:* `T` shall be *Cpp17MoveInsertable* and *Cpp17DefaultInsertable* into `*this`.~~

17 *Remarks:* If an exception is thrown other than by the move constructor of a non-*Cpp17CopyInsertable* `T` there are no effects.

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

18 *Expects:* `T` shall be *Cpp17CopyInsertable* into `*this`.

19 *Effects:* If `sz < size()`, erases the last `size() - sz` elements from the sequence. Otherwise, appends `sz - size()` copies of `c` to the sequence.

20 ~~*Requires:* `T` shall be *Cpp17CopyInsertable* into `*this`.~~

21 *Remarks:* If an exception is thrown there are no effects.

230) `reserve()` uses `Allocator::allocate()` which may throw an appropriate exception.

21.3.11.4 Data [vector.data]

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

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

2 *Complexity:* Constant time.

21.3.11.5 Modifiers [vector.modifiers]

```
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template<class InputIterator>
  iterator insert(const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<T>);

template<class... Args> reference emplace_back(Args&&... args);
template<class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_back(const T& x);
void push_back(T&& x);
```

1 *Remarks:* Causes reallocation if the new size is greater than the old capacity. Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. 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, move constructor, assignment operator, or move assignment operator of `T` or by any `InputIterator` operation there are no effects. If an exception is thrown while inserting a single element at the end and `T` is *Cpp17CopyInsertable* or `is_nothrow_move_constructible_v<T>` is `true`, there are no effects. Otherwise, if an exception is thrown by the move constructor of a non-*Cpp17CopyInsertable* `T`, the effects are unspecified.

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

```
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void pop_back();
```

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

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

5 *Throws:* Nothing unless an exception is thrown by the assignment operator or move assignment operator of `T`.

21.3.11.6 Erasure [vector.erasure]

```
template <class T, class Allocator, class U>
  void erase(vector<T, Allocator>& c, const U& value);
```

1 *Effects:* Equivalent to: `c.erase(remove(c.begin(), c.end(), value), c.end());`

```
template <class T, class Allocator, class Predicate>
  void erase_if(vector<T, Allocator>& c, Predicate pred);
```

2 *Effects:* Equivalent to: `c.erase(remove_if(c.begin(), c.end(), pred), c.end());`

21.3.12 Class `vector<bool>` [vector.bool]

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

```
namespace std {
  template<class Allocator>
  class vector<bool, Allocator> {
  public:
    // types
```

```
using value_type             = bool;
using allocator_type         = Allocator;
using pointer                = implementation-defined;
using const_pointer          = implementation-defined;
using const_reference        = bool;
using size_type              = implementation-defined; // see 21.2
using difference_type        = implementation-defined; // see 21.2
using iterator               = implementation-defined; // see 21.2
using const_iterator         = implementation-defined; // see 21.2
using reverse_iterator       = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;

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

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

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

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

// capacity
[[nodiscard]] bool empty() const noexcept;
```

```
    size_type size() const noexcept;
    size_type max_size() const noexcept;
    size_type capacity() const noexcept;
    void      resize(size_type sz, bool c = false);
    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
    template<class... Args> reference emplace_back(Args&&... args);
    void push_back(const bool& x);
    void pop_back();
    template<class... Args> iterator emplace(const_iterator position, Args&&... args);
    iterator insert(const_iterator position, const bool& x);
    iterator insert(const_iterator position, size_type n, const bool& x);
    template<class InputIterator>
      iterator insert(const_iterator position, InputIterator first, InputIterator last);
    iterator insert(const_iterator position, initializer_list<bool> il);

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

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 and `allocator_traits::construct` (**??**) is not used to construct these values.

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 function 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() noexcept;`

5 *Effects:* Replaces each element in the container with its complement.

`static void swap(reference x, reference y) noexcept;`

6 *Effects:* Exchanges the contents of `x` and `y` as if by:

```
bool b = x;
x = y;
y = b;
```

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

7 The specialization is enabled (**??**).

21.4 Associative containers [associative]

21.4.1 In general [associative.general]

1 The header `<map>` defines the class templates `map` and `multimap`; the header `<set>` defines the class templates `set` and `multiset`.

2 The following exposition-only alias templates may appear in deduction guides for associative containers:

```
template<class InputIterator>
  using iter-value-type =
    typename iterator_traits<InputIterator>::value_type;                // exposition only
template<class InputIterator>
  using iter-key-type = remove_const_t<
    typename iterator_traits<InputIterator>::value_type::first_type>;   // exposition only
template<class InputIterator>
  using iter-mapped-type =
    typename iterator_traits<InputIterator>::value_type::second_type;   // exposition only
template<class InputIterator>
  using iter-to-alloc-type = pair<
    add_const_t<typename iterator_traits<InputIterator>::value_type::first_type>,
    typename iterator_traits<InputIterator>::value_type::second_type>;  // exposition only
```

21.4.2 Header <map> synopsis [associative.map.syn]

```
#include <initializer_list>

namespace std {
  // 21.4.4, class template map
  template<class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T>>>
    class map;

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

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

  template <class Key, class T, class Compare, class Allocator, class Predicate>
    void erase_if(map<Key, T, Compare, Allocator>& c, Predicate pred);

  // 21.4.5, class template multimap
  template<class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T>>>
    class multimap;
```

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

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

  template <class Key, class T, class Compare, class Allocator, class Predicate>
    void erase_if(multimap<Key, T, Compare, Allocator>& c, Predicate pred);

  namespace pmr {
    template<class Key, class T, class Compare = less<Key>>
      using map = std::map<Key, T, Compare,
                           polymorphic_allocator<pair<const Key, T>>>;

    template<class Key, class T, class Compare = less<Key>>
      using multimap = std::multimap<Key, T, Compare,
                                     polymorphic_allocator<pair<const Key, T>>>;
  }
}
```

21.4.3 Header <set> synopsis [associative.set.syn]

```
#include <initializer_list>

namespace std {
  // 21.4.6, class template set
  template<class Key, class Compare = less<Key>, class Allocator = allocator<Key>>
    class set;

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

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

  template <class Key, class Compare, class Allocator, class Predicate>
    void erase_if(set<Key, Compare, Allocator>& c, Predicate pred);

  // 21.4.7, class template multiset
  template<class Key, class Compare = less<Key>, class Allocator = allocator<Key>>
    class multiset;

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

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

  template <class Key, class Compare, class Allocator, class Predicate>
    void erase_if(multiset<Key, Compare, Allocator>& c, Predicate pred);

  namespace pmr {
    template<class Key, class Compare = less<Key>>
      using set = std::set<Key, Compare, polymorphic_allocator<Key>>;

    template<class Key, class Compare = less<Key>>
      using multiset = std::multiset<Key, Compare, polymorphic_allocator<Key>>;
  }
}
```

21.4.4 Class template `map` [map]

21.4.4.1 Overview [map.overview]

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 (21.2), of an associative container (21.2.6), and of an allocator-aware container (Table 65). A `map` also provides most operations described in 21.2.6 for unique keys. This means that a `map` supports the `a_uniq` operations in 21.2.6 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 Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T>>>
  class map {
  public:
    // types
    using key_type               = Key;
    using mapped_type            = T;
    using value_type             = pair<const Key, T>;
    using key_compare            = Compare;
    using allocator_type         = Allocator;
    using pointer                = typename allocator_traits<Allocator>::pointer;
    using const_pointer          = typename allocator_traits<Allocator>::const_pointer;
    using reference              = value_type&;
    using const_reference        = const value_type&;
    using size_type              = implementation-defined; // see 21.2
    using difference_type        = implementation-defined; // see 21.2
    using iterator               = implementation-defined; // see 21.2
    using const_iterator         = implementation-defined; // see 21.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    using node_type              = unspecified;
    using insert_return_type     = insert-return-type<iterator, node_type>;

    class value_compare {
      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);
      }
    };

    // 21.4.4.2, construct/copy/destroy
    map() : map(Compare()) { }
    explicit map(const Compare& comp, const Allocator& = Allocator());
    template<class InputIterator>
      map(InputIterator first, InputIterator last,
          const Compare& comp = Compare(), const Allocator& = Allocator());
    map(const map& x);
    map(map&& x);
    explicit map(const Allocator&);
    map(const map&, const Allocator&);
    map(map&&, const Allocator&);
    map(initializer_list<value_type>,
      const Compare& = Compare(),
      const Allocator& = Allocator());
    template<class InputIterator>
      map(InputIterator first, InputIterator last, const Allocator& a)
        : map(first, last, Compare(), a) { }
    map(initializer_list<value_type> il, const Allocator& a)
      : map(il, Compare(), a) { }
    ~map();
    map& operator=(const map& x);
    map& operator=(map&& x)
      noexcept(allocator_traits<Allocator>::is_always_equal::value &&
               is_nothrow_move_assignable_v<Compare>);
    map& operator=(initializer_list<value_type>);
    allocator_type get_allocator() const noexcept;

    // iterators
    iterator               begin() noexcept;
```

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

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

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

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// 21.4.4.3, element access
mapped_type& operator[](const key_type& x);
mapped_type& operator[](key_type&& x);
mapped_type&       at(const key_type& x);
const mapped_type& at(const key_type& x) const;

// 21.4.4.4, modifiers
template<class... Args> pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& x);
pair<iterator, bool> insert(value_type&& x);
template<class P> pair<iterator, bool> insert(P&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template<class P>
  iterator insert(const_iterator position, P&&);
template<class InputIterator>
  void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
insert_return_type insert(node_type&& nh);
iterator           insert(const_iterator hint, node_type&& nh);

template<class... Args>
  pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
  pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template<class M>
  pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
  pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class M>
  iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
template<class M>
  iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);

iterator  erase(iterator position);
iterator  erase(const_iterator position);
size_type erase(const key_type& x);
```

```
  iterator  erase(const_iterator first, const_iterator last);
  void      swap(map&)
    noexcept(allocator_traits<Allocator>::is_always_equal::value &&
             is_nothrow_swappable_v<Compare>);
  void      clear() noexcept;

  template<class C2>
    void merge(map<Key, T, C2, Allocator>& source);
  template<class C2>
    void merge(map<Key, T, C2, Allocator>&& source);
  template<class C2>
    void merge(multimap<Key, T, C2, Allocator>& source);
  template<class C2>
    void merge(multimap<Key, T, C2, Allocator>&& source);

  // 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;
  template<class K> iterator       find(const K& x);
  template<class K> const_iterator find(const K& x) const;

  size_type      count(const key_type& x) const;
  template<class K> size_type count(const K& x) const;

  bool           contains(const key_type& x) const;
  template<class K> bool contains(const K& x) const;

  iterator       lower_bound(const key_type& x);
  const_iterator lower_bound(const key_type& x) const;
  template<class K> iterator       lower_bound(const K& x);
  template<class K> const_iterator lower_bound(const K& x) const;

  iterator       upper_bound(const key_type& x);
  const_iterator upper_bound(const key_type& x) const;
  template<class K> iterator       upper_bound(const K& x);
  template<class K> const_iterator upper_bound(const K& 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 K>
    pair<iterator, iterator>             equal_range(const K& x);
  template<class K>
    pair<const_iterator, const_iterator> equal_range(const K& x) const;
};

template<class InputIterator, class Compare = less<iter-key-type<InputIterator>>,
         class Allocator = allocator<iter-to-alloc-type<InputIterator>>>
  map(InputIterator, InputIterator, Compare = Compare(), Allocator = Allocator())
    -> map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Compare, Allocator>;

template<class Key, class T, class Compare = less<Key>,
         class Allocator = allocator<pair<const Key, T>>>
  map(initializer_list<pair<Key, T>>, Compare = Compare(), Allocator = Allocator())
    -> map<Key, T, Compare, Allocator>;

template<class InputIterator, class Allocator>
  map(InputIterator, InputIterator, Allocator)
    -> map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
           less<iter-key-type<InputIterator>>, Allocator>;
```

```
  template<class Key, class T, class Allocator>
    map(initializer_list<pair<Key, T>>, Allocator) -> map<Key, T, less<Key>, Allocator>;

  // swap
  template<class Key, class T, class Compare, class Allocator>
    void swap(map<Key, T, Compare, Allocator>& x,
              map<Key, T, Compare, Allocator>& y)
      noexcept(noexcept(x.swap(y)));
}
```

21.4.4.2 Constructors, copy, and assignment [map.cons]

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

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

2 *Complexity:* Constant.

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

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

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

21.4.4.3 Element access [map.access]

```
mapped_type& operator[](const key_type& x);
```

1 *Effects:* Equivalent to: `return try_emplace(x).first->second;`

```
mapped_type& operator[](key_type&& x);
```

2 *Effects:* Equivalent to: `return try_emplace(move(x)).first->second;`

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

3 *Returns:* A reference to the `mapped_type` corresponding to `x` in `*this`.

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

5 *Complexity:* Logarithmic.

21.4.4.4 Modifiers [map.modifiers]

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

1 *Constraints:* `is_constructible_v<value_type, P&&>` is `true`.

2 *Effects:* The first form is equivalent to `return emplace(std::forward<P>(x))`. The second form is equivalent to `return emplace_hint(position, std::forward<P>(x))`.

3 ~~*Remarks:* These signatures shall not participate in overload resolution unless `is_constructible_v<value_type, P&&>` is `true`.~~

```
template<class... Args>
  pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
```

4 ~~*Requires:*~~ *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `map` from `piecewise_construct`, `forward_as_tuple(k)`, `forward_as_tuple(std::forward<Args>(args)...)`.

5 *Effects:* If the map already contains an element whose key is equivalent to `k`, there is no effect. Otherwise inserts an object of type `value_type` constructed with `piecewise_construct`, `forward_as_tuple(k)`, `forward_as_tuple(std::forward<Args>(args)...)`.

6 *Returns:* In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to `k`.

7 *Complexity:* The same as `emplace` and `emplace_hint`, respectively.

```
template<class... Args>
  pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
```

8 ~~*Requires:*~~ *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `map` from `piecewise_construct`, `forward_as_tuple(std::move(k))`, `forward_as_tuple(std::forward<Args>(args)...)`.

9 *Effects:* If the map already contains an element whose key is equivalent to `k`, there is no effect. Otherwise inserts an object of type `value_type` constructed with `piecewise_construct`, `forward_as_tuple(std::move(k))`, `forward_as_tuple(std::forward<Args>(args)...)`.

10 *Returns:* In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to `k`.

11 *Complexity:* The same as `emplace` and `emplace_hint`, respectively.

```
template<class M>
  pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
  iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
```

12 ~~*Requires:*~~ *Mandates:* `is_assignable_v<mapped_type&, M&&>` shall be `true`.

13 *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `map` from `k`, `forward<M>(obj)`.

14 *Effects:* If the map already contains an element `e` whose key is equivalent to `k`, assigns `std::forward<M>(obj)` to `e.second`. Otherwise inserts an object of type `value_type` constructed with `k`, `std::forward<M>(obj)`.

15 *Returns:* In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to `k`.

16 *Complexity:* The same as `emplace` and `emplace_hint`, respectively.

```
template<class M>
  pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class M>
  iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);
```

17 ~~*Requires:*~~ *Mandates:* `is_assignable_v<mapped_type&, M&&>` shall be `true`.

18 *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `map` from `move(k)`, `forward<M>(obj)`.

19 *Effects:* If the map already contains an element `e` whose key is equivalent to `k`, assigns `std::forward<M>(obj)` to `e.second`. Otherwise inserts an object of type `value_type` constructed with `std::move(k)`, `std::forward<M>(obj)`.

20 *Returns:* In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to `k`.

21 *Complexity:* The same as `emplace` and `emplace_hint`, respectively.

21.4.4.5 Erasure [map.erasure]

```
template <class Key, class T, class Compare, class Allocator, class Predicate>
  void erase_if(map<Key, T, Compare, Allocator>& c, Predicate pred);
```

1 *Effects:* Equivalent to:

```
for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
  } else {
    ++i;
  }
}
```

21.4.5 Class template `multimap` [multimap]

21.4.5.1 Overview [multimap.overview]

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 (21.2), of an associative container (21.2.6), and of an allocator-aware container (Table 65). A `multimap` also provides most operations described in 21.2.6 for equal keys. This means that a `multimap` supports the `a_eq` operations in 21.2.6 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<class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T>>>
  class multimap {
  public:
    // types
    using key_type               = Key;
    using mapped_type            = T;
    using value_type             = pair<const Key, T>;
    using key_compare            = Compare;
    using allocator_type         = Allocator;
    using pointer                = typename allocator_traits<Allocator>::pointer;
    using const_pointer          = typename allocator_traits<Allocator>::const_pointer;
    using reference              = value_type&;
    using const_reference        = const value_type&;
    using size_type              = implementation-defined; // see 21.2
    using difference_type        = implementation-defined; // see 21.2
    using iterator               = implementation-defined; // see 21.2
    using const_iterator         = implementation-defined; // see 21.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    using node_type              = unspecified;

    class value_compare {
      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);
      }
    };

    // 21.4.5.2, construct/copy/destroy
    multimap() : multimap(Compare()) { }
    explicit multimap(const Compare& comp, const Allocator& = Allocator());
    template<class InputIterator>
      multimap(InputIterator first, InputIterator last,
               const Compare& comp = Compare(),
               const Allocator& = Allocator());
    multimap(const multimap& x);
    multimap(multimap&& x);
    explicit multimap(const Allocator&);
    multimap(const multimap&, const Allocator&);
    multimap(multimap&&, const Allocator&);
    multimap(initializer_list<value_type>,
      const Compare& = Compare(),
      const Allocator& = Allocator());
```

```
template<class InputIterator>
  multimap(InputIterator first, InputIterator last, const Allocator& a)
    : multimap(first, last, Compare(), a) { }
multimap(initializer_list<value_type> il, const Allocator& a)
  : multimap(il, Compare(), a) { }
~multimap();
multimap& operator=(const multimap& x);
multimap& operator=(multimap&& x)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_move_assignable_v<Compare>);
multimap& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

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

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

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

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// 21.4.5.3, modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& x);
iterator insert(value_type&& x);
template<class P> iterator insert(P&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template<class P> iterator insert(const_iterator position, P&& x);
template<class InputIterator>
  void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

iterator  erase(iterator position);
iterator  erase(const_iterator position);
size_type erase(const key_type& x);
iterator  erase(const_iterator first, const_iterator last);
void      swap(multimap&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_swappable_v<Compare>);
void      clear() noexcept;

template<class C2>
  void merge(multimap<Key, T, C2, Allocator>& source);
```

```
  template<class C2>
    void merge(multimap<Key, T, C2, Allocator>&& source);
  template<class C2>
    void merge(map<Key, T, C2, Allocator>& source);
  template<class C2>
    void merge(map<Key, T, C2, Allocator>&& source);

  // 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;
  template<class K> iterator       find(const K& x);
  template<class K> const_iterator find(const K& x) const;

  size_type      count(const key_type& x) const;
  template<class K> size_type count(const K& x) const;

  bool           contains(const key_type& x) const;
  template<class K> bool contains(const K& x) const;

  iterator       lower_bound(const key_type& x);
  const_iterator lower_bound(const key_type& x) const;
  template<class K> iterator       lower_bound(const K& x);
  template<class K> const_iterator lower_bound(const K& x) const;

  iterator       upper_bound(const key_type& x);
  const_iterator upper_bound(const key_type& x) const;
  template<class K> iterator       upper_bound(const K& x);
  template<class K> const_iterator upper_bound(const K& 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 K>
    pair<iterator, iterator>             equal_range(const K& x);
  template<class K>
    pair<const_iterator, const_iterator> equal_range(const K& x) const;
};

template<class InputIterator, class Compare = less<iter-key-type<InputIterator>>,
         class Allocator = allocator<iter-to-alloc-type<InputIterator>>>
  multimap(InputIterator, InputIterator, Compare = Compare(), Allocator = Allocator())
    -> multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                Compare, Allocator>;

template<class Key, class T, class Compare = less<Key>,
         class Allocator = allocator<pair<const Key, T>>>
  multimap(initializer_list<pair<Key, T>>, Compare = Compare(), Allocator = Allocator())
    -> multimap<Key, T, Compare, Allocator>;

template<class InputIterator, class Allocator>
  multimap(InputIterator, InputIterator, Allocator)
    -> multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                less<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
  multimap(initializer_list<pair<Key, T>>, Allocator)
    -> multimap<Key, T, less<Key>, Allocator>;
```

```
  // swap
  template<class Key, class T, class Compare, class Allocator>
    void swap(multimap<Key, T, Compare, Allocator>& x,
              multimap<Key, T, Compare, Allocator>& y)
      noexcept(noexcept(x.swap(y)));
}
```

21.4.5.2 Constructors [multimap.cons]

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

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

2 *Complexity:* Constant.

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

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

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

21.4.5.3 Modifiers [multimap.modifiers]

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

1 *Constraints:* `is_constructible_v<value_type, P&&>` is `true`.

2 *Effects:* The first form is equivalent to `return emplace(std::forward<P>(x))`. The second form is equivalent to `return emplace_hint(position, std::forward<P>(x))`.

3 ~~*Remarks:* These signatures shall not participate in overload resolution unless `is_constructible_v<value_type, P&&>` is `true`.~~

21.4.5.4 Erasure [multimap.erasure]

```
template <class Key, class T, class Compare, class Allocator, class Predicate>
  void erase_if(multimap<Key, T, Compare, Allocator>& c, Predicate pred);
```

1 *Effects:* Equivalent to:

```
for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
  } else {
    ++i;
  }
}
```

21.4.6 Class template set [set]

21.4.6.1 Overview [set.overview]

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. The `set` class supports bidirectional iterators.

2 A `set` satisfies all of the requirements of a container, of a reversible container (21.2), of an associative container (21.2.6), and of an allocator-aware container (Table 65). A `set` also provides most operations described in 21.2.6 for unique keys. This means that a `set` supports the `a_uniq` operations in 21.2.6 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<class Key, class Compare = less<Key>,
           class Allocator = allocator<Key>>
  class set {
  public:
    // types
    using key_type               = Key;
    using key_compare            = Compare;
    using value_type             = Key;
    using value_compare          = Compare;
    using allocator_type         = Allocator;
    using pointer                = typename allocator_traits<Allocator>::pointer;
    using const_pointer          = typename allocator_traits<Allocator>::const_pointer;
    using reference              = value_type&;
    using const_reference        = const value_type&;
    using size_type              = implementation-defined; // see 21.2
    using difference_type        = implementation-defined; // see 21.2
    using iterator               = implementation-defined; // see 21.2
    using const_iterator         = implementation-defined; // see 21.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    using node_type              = unspecified;
    using insert_return_type     = insert-return-type<iterator, node_type>;

    // 21.4.6.2, construct/copy/destroy
    set() : set(Compare()) { }
    explicit set(const Compare& comp, const Allocator& = Allocator());
    template<class InputIterator>
      set(InputIterator first, InputIterator last,
          const Compare& comp = Compare(), const Allocator& = Allocator());
    set(const set& x);
    set(set&& x);
    explicit set(const Allocator&);
    set(const set&, const Allocator&);
    set(set&&, const Allocator&);
    set(initializer_list<value_type>, const Compare& = Compare(),
        const Allocator& = Allocator());
    template<class InputIterator>
      set(InputIterator first, InputIterator last, const Allocator& a)
        : set(first, last, Compare(), a) { }
    set(initializer_list<value_type> il, const Allocator& a)
      : set(il, Compare(), a) { }
    ~set();
    set& operator=(const set& x);
    set& operator=(set&& x)
      noexcept(allocator_traits<Allocator>::is_always_equal::value &&
               is_nothrow_move_assignable_v<Compare>);
    set& operator=(initializer_list<value_type>);
    allocator_type get_allocator() const noexcept;

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

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

    const_iterator         cbegin() const noexcept;
    const_iterator         cend() const noexcept;
    const_reverse_iterator crbegin() const noexcept;
```

```
const_reverse_iterator crend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator,bool> insert(const value_type& x);
pair<iterator,bool> insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template<class InputIterator>
  void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
insert_return_type insert(node_type&& nh);
iterator           insert(const_iterator hint, node_type&& nh);

iterator  erase(iterator position);
iterator  erase(const_iterator position);
size_type erase(const key_type& x);
iterator  erase(const_iterator first, const_iterator last);
void      swap(set&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_swappable_v<Compare>);
void      clear() noexcept;

template<class C2>
  void merge(set<Key, C2, Allocator>& source);
template<class C2>
  void merge(set<Key, C2, Allocator>&& source);
template<class C2>
  void merge(multiset<Key, C2, Allocator>& source);
template<class C2>
  void merge(multiset<Key, C2, Allocator>&& source);

// 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;
template<class K> iterator       find(const K& x);
template<class K> const_iterator find(const K& x) const;

size_type      count(const key_type& x) const;
template<class K> size_type count(const K& x) const;

bool           contains(const key_type& x) const;
template<class K> bool contains(const K& x) const;

iterator       lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
template<class K> iterator       lower_bound(const K& x);
template<class K> const_iterator lower_bound(const K& x) const;

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

```
    template<class K> iterator       upper_bound(const K& x);
    template<class K> const_iterator upper_bound(const K& 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 K>
      pair<iterator, iterator>             equal_range(const K& x);
    template<class K>
      pair<const_iterator, const_iterator> equal_range(const K& x) const;
  };

  template<class InputIterator,
           class Compare = less<iter-value-type<InputIterator>>,
           class Allocator = allocator<iter-value-type<InputIterator>>>
    set(InputIterator, InputIterator,
        Compare = Compare(), Allocator = Allocator())
      -> set<iter-value-type<InputIterator>, Compare, Allocator>;

  template<class Key, class Compare = less<Key>, class Allocator = allocator<Key>>
    set(initializer_list<Key>, Compare = Compare(), Allocator = Allocator())
      -> set<Key, Compare, Allocator>;

  template<class InputIterator, class Allocator>
    set(InputIterator, InputIterator, Allocator)
      -> set<iter-value-type<InputIterator>,
             less<iter-value-type<InputIterator>>, Allocator>;

  template<class Key, class Allocator>
    set(initializer_list<Key>, Allocator) -> set<Key, less<Key>, Allocator>;

  // swap
  template<class Key, class Compare, class Allocator>
    void swap(set<Key, Compare, Allocator>& x,
              set<Key, Compare, Allocator>& y)
      noexcept(noexcept(x.swap(y)));
}
```

21.4.6.2 Constructors, copy, and assignment [set.cons]

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

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

2 *Complexity:* Constant.

```
template<class InputIterator>
  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 *Complexity:* Linear in N if the range `[first, last)` is already sorted using `comp` and otherwise $N \log N$, where N is `last - first`.

21.4.6.3 Erasure [set.erasure]

```
template <class Key, class Compare, class Allocator, class Predicate>
  void erase_if(set<Key, Compare, Allocator>& c, Predicate pred);
```

1 *Effects:* Equivalent to:

```
for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
  } else {
    ++i;
```

```
    }
  }
```

21.4.7 Class template `multiset` [multiset]

21.4.7.1 Overview [multiset.overview]

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. The `multiset` class supports bidirectional iterators.

2 A `multiset` satisfies all of the requirements of a container, of a reversible container (21.2), of an associative container (21.2.6), and of an allocator-aware container (Table 65). `multiset` also provides most operations described in 21.2.6 for duplicate keys. This means that a `multiset` supports the `a_eq` operations in 21.2.6 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 Key, class Compare = less<Key>,
           class Allocator = allocator<Key>>
  class multiset {
  public:
    // types
    using key_type               = Key;
    using key_compare            = Compare;
    using value_type             = Key;
    using value_compare          = Compare;
    using allocator_type         = Allocator;
    using pointer                = typename allocator_traits<Allocator>::pointer;
    using const_pointer          = typename allocator_traits<Allocator>::const_pointer;
    using reference              = value_type&;
    using const_reference        = const value_type&;
    using size_type              = implementation-defined; // see 21.2
    using difference_type        = implementation-defined; // see 21.2
    using iterator               = implementation-defined; // see 21.2
    using const_iterator         = implementation-defined; // see 21.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    using node_type              = unspecified;

    // 21.4.7.2, construct/copy/destroy
    multiset() : multiset(Compare()) { }
    explicit multiset(const Compare& comp, const Allocator& = Allocator());
    template<class InputIterator>
      multiset(InputIterator first, InputIterator last,
               const Compare& comp = Compare(), const Allocator& = Allocator());
    multiset(const multiset& x);
    multiset(multiset&& x);
    explicit multiset(const Allocator&);
    multiset(const multiset&, const Allocator&);
    multiset(multiset&&, const Allocator&);
    multiset(initializer_list<value_type>, const Compare& = Compare(),
             const Allocator& = Allocator());
    template<class InputIterator>
      multiset(InputIterator first, InputIterator last, const Allocator& a)
        : multiset(first, last, Compare(), a) { }
    multiset(initializer_list<value_type> il, const Allocator& a)
      : multiset(il, Compare(), a) { }
    ~multiset();
    multiset& operator=(const multiset& x);
    multiset& operator=(multiset&& x)
      noexcept(allocator_traits<Allocator>::is_always_equal::value &&
               is_nothrow_move_assignable_v<Compare>);
    multiset& operator=(initializer_list<value_type>);
    allocator_type get_allocator() const noexcept;
```

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

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

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

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& x);
iterator insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template<class InputIterator>
  void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

iterator  erase(iterator position);
iterator  erase(const_iterator position);
size_type erase(const key_type& x);
iterator  erase(const_iterator first, const_iterator last);
void      swap(multiset&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_swappable_v<Compare>);
void      clear() noexcept;

template<class C2>
  void merge(multiset<Key, C2, Allocator>& source);
template<class C2>
  void merge(multiset<Key, C2, Allocator>&& source);
template<class C2>
  void merge(set<Key, C2, Allocator>& source);
template<class C2>
  void merge(set<Key, C2, Allocator>&& source);

// 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;
template<class K> iterator       find(const K& x);
template<class K> const_iterator find(const K& x) const;
```

```
    size_type      count(const key_type& x) const;
    template<class K> size_type count(const K& x) const;

    bool           contains(const key_type& x) const;
    template<class K> bool contains(const K& x) const;

    iterator       lower_bound(const key_type& x);
    const_iterator lower_bound(const key_type& x) const;
    template<class K> iterator       lower_bound(const K& x);
    template<class K> const_iterator lower_bound(const K& x) const;

    iterator       upper_bound(const key_type& x);
    const_iterator upper_bound(const key_type& x) const;
    template<class K> iterator       upper_bound(const K& x);
    template<class K> const_iterator upper_bound(const K& 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 K>
      pair<iterator, iterator>             equal_range(const K& x);
    template<class K>
      pair<const_iterator, const_iterator> equal_range(const K& x) const;
  };

  template<class InputIterator,
           class Compare = less<iter-value-type<InputIterator>>,
           class Allocator = allocator<iter-value-type<InputIterator>>>
    multiset(InputIterator, InputIterator,
             Compare = Compare(), Allocator = Allocator())
      -> multiset<iter-value-type<InputIterator>, Compare, Allocator>;

  template<class Key, class Compare = less<Key>, class Allocator = allocator<Key>>
    multiset(initializer_list<Key>, Compare = Compare(), Allocator = Allocator())
      -> multiset<Key, Compare, Allocator>;

  template<class InputIterator, class Allocator>
    multiset(InputIterator, InputIterator, Allocator)
      -> multiset<iter-value-type<InputIterator>,
                  less<iter-value-type<InputIterator>>, Allocator>;

  template<class Key, class Allocator>
    multiset(initializer_list<Key>, Allocator) -> multiset<Key, less<Key>, Allocator>;

  // swap
  template<class Key, class Compare, class Allocator>
    void swap(multiset<Key, Compare, Allocator>& x,
              multiset<Key, Compare, Allocator>& y)
      noexcept(noexcept(x.swap(y)));
}
```

21.4.7.2 Constructors [multiset.cons]

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

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

2 *Complexity:* Constant.

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

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

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

21.4.7.3 Erasure [multiset.erasure]

```
template <class Key, class Compare, class Allocator, class Predicate>
  void erase_if(multiset<Key, Compare, Allocator>& c, Predicate pred);
```

1 *Effects:* Equivalent to:

```
for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
  } else {
    ++i;
  }
}
```

21.5 Unordered associative containers [unord]

21.5.1 In general [unord.general]

1 The header `<unordered_map>` defines the class templates `unordered_map` and `unordered_multimap`; the header `<unordered_set>` defines the class templates `unordered_set` and `unordered_multiset`.

2 The exposition-only alias templates *iter-value-type*, *iter-key-type*, *iter-mapped-type*, and *iter-to-alloc-type* defined in 21.4.1 may appear in deduction guides for unordered containers.

21.5.2 Header `<unordered_map>` synopsis [unord.map.syn]

```
#include <initializer_list>

namespace std {
  // 21.5.4, class template unordered_map
  template<class Key,
           class T,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Alloc = allocator<pair<const Key, T>>>
    class unordered_map;

  // 21.5.5, class template unordered_multimap
  template<class Key,
           class T,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Alloc = allocator<pair<const Key, T>>>
    class unordered_multimap;

  template<class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_map<Key, T, Hash, Pred, Alloc>& b);
  template<class Key, class T, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_map<Key, T, Hash, Pred, Alloc>& b);

  template<class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
  template<class Key, class T, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);

  template<class Key, class 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)
      noexcept(noexcept(x.swap(y)));
```

```
  template<class Key, class 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)
      noexcept(noexcept(x.swap(y)));

  template <class K, class T, class H, class P, class A, class Predicate>
    void erase_if(unordered_map<K, T, H, P, A>& c, Predicate pred);

  template <class K, class T, class H, class P, class A, class Predicate>
    void erase_if(unordered_multimap<K, T, H, P, A>& c, Predicate pred);

  namespace pmr {
    template<class Key,
             class T,
             class Hash = hash<Key>,
             class Pred = equal_to<Key>>
      using unordered_map =
        std::unordered_map<Key, T, Hash, Pred,
                           polymorphic_allocator<pair<const Key, T>>>;
    template<class Key,
             class T,
             class Hash = hash<Key>,
             class Pred = equal_to<Key>>
      using unordered_multimap =
        std::unordered_multimap<Key, T, Hash, Pred,
                                polymorphic_allocator<pair<const Key, T>>>;

  }
}
```

21.5.3 Header <unordered_set> synopsis [unord.set.syn]

```
#include <initializer_list>

namespace std {
  // 21.5.6, class template unordered_set
  template<class Key,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Alloc = allocator<Key>>
    class unordered_set;

  // 21.5.7, class template unordered_multiset
  template<class Key,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Alloc = allocator<Key>>
    class unordered_multiset;

  template<class Key, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_set<Key, Hash, Pred, Alloc>& a,
                    const unordered_set<Key, Hash, Pred, Alloc>& b);
  template<class Key, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_set<Key, Hash, Pred, Alloc>& a,
                    const unordered_set<Key, Hash, Pred, Alloc>& b);

  template<class Key, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_multiset<Key, Hash, Pred, Alloc>& a,
                    const unordered_multiset<Key, Hash, Pred, Alloc>& b);
  template<class Key, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_multiset<Key, Hash, Pred, Alloc>& a,
                    const unordered_multiset<Key, Hash, Pred, Alloc>& b);
```

```
  template<class Key, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
              unordered_set<Key, Hash, Pred, Alloc>& y)
      noexcept(noexcept(x.swap(y)));

  template<class Key, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
              unordered_multiset<Key, Hash, Pred, Alloc>& y)
      noexcept(noexcept(x.swap(y)));

  template <class K, class H, class P, class A, class Predicate>
    void erase_if(unordered_set<K, H, P, A>& c, Predicate pred);

  template <class K, class H, class P, class A, class Predicate>
    void erase_if(unordered_multiset<K, H, P, A>& c, Predicate pred);

  namespace pmr {
    template<class Key,
             class Hash = hash<Key>,
             class Pred = equal_to<Key>>
      using unordered_set = std::unordered_set<Key, Hash, Pred,
                                               polymorphic_allocator<Key>>;

    template<class Key,
             class Hash = hash<Key>,
             class Pred = equal_to<Key>>
      using unordered_multiset = std::unordered_multiset<Key, Hash, Pred,
                                                         polymorphic_allocator<Key>>;
  }
}
```

21.5.4 Class template `unordered_map` [unord.map]

21.5.4.1 Overview [unord.map.overview]

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. The `unordered_map` class supports forward iterators.

2 An `unordered_map` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 65). 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 subclause 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<class Key,
           class T,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Allocator = allocator<pair<const Key, T>>>
  class unordered_map {
  public:
    // types
    using key_type             = Key;
    using mapped_type          = T;
    using value_type           = pair<const Key, T>;
    using hasher               = Hash;
    using key_equal            = see 21.2.7;
    using allocator_type       = Allocator;
    using pointer              = typename allocator_traits<Allocator>::pointer;
    using const_pointer        = typename allocator_traits<Allocator>::const_pointer;
    using reference            = value_type&;
```

```
using const_reference      = const value_type&;
using size_type            = implementation-defined; // see 21.2
using difference_type      = implementation-defined; // see 21.2

using iterator             = implementation-defined; // see 21.2
using const_iterator       = implementation-defined; // see 21.2
using local_iterator       = implementation-defined; // see 21.2
using const_local_iterator = implementation-defined; // see 21.2
using node_type            = unspecified;
using insert_return_type   = insert-return-type<iterator, node_type>;

// 21.5.4.2, construct/copy/destroy
unordered_map();
explicit unordered_map(size_type n,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
template<class InputIterator>
  unordered_map(InputIterator f, InputIterator l,
                size_type n = see below,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
unordered_map(const unordered_map&);
unordered_map(unordered_map&&);
explicit unordered_map(const Allocator&);
unordered_map(const unordered_map&, const Allocator&);
unordered_map(unordered_map&&, const Allocator&);
unordered_map(initializer_list<value_type> il,
              size_type n = see below,
              const hasher& hf = hasher(),
              const key_equal& eql = key_equal(),
              const allocator_type& a = allocator_type());
unordered_map(size_type n, const allocator_type& a)
  : unordered_map(n, hasher(), key_equal(), a) { }
unordered_map(size_type n, const hasher& hf, const allocator_type& a)
  : unordered_map(n, hf, key_equal(), a) { }
template<class InputIterator>
  unordered_map(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
    : unordered_map(f, l, n, hasher(), key_equal(), a) { }
template<class InputIterator>
  unordered_map(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                const allocator_type& a)
    : unordered_map(f, l, n, hf, key_equal(), a) { }
unordered_map(initializer_list<value_type> il, size_type n, const allocator_type& a)
  : unordered_map(il, n, hasher(), key_equal(), a) { }
unordered_map(initializer_list<value_type> il, size_type n, const hasher& hf,
              const allocator_type& a)
  : unordered_map(il, n, hf, key_equal(), a) { }
~unordered_map();
unordered_map& operator=(const unordered_map&);
unordered_map& operator=(unordered_map&&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_move_assignable_v<Hash> &&
           is_nothrow_move_assignable_v<Pred>);
unordered_map& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator       begin() noexcept;
const_iterator begin() const noexcept;
iterator       end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
```

```
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// 21.5.4.4, modifiers
template<class... Args> pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& obj);
pair<iterator, bool> insert(value_type&& obj);
template<class P> pair<iterator, bool> insert(P&& obj);
iterator       insert(const_iterator hint, const value_type& obj);
iterator       insert(const_iterator hint, value_type&& obj);
template<class P> iterator insert(const_iterator hint, P&& obj);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
insert_return_type insert(node_type&& nh);
iterator           insert(const_iterator hint, node_type&& nh);

template<class... Args>
  pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
  pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template<class M>
  pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
  pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class M>
  iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
template<class M>
  iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);

iterator  erase(iterator position);
iterator  erase(const_iterator position);
size_type erase(const key_type& k);
iterator  erase(const_iterator first, const_iterator last);
void      swap(unordered_map&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_swappable_v<Hash> &&
           is_nothrow_swappable_v<Pred>);
void      clear() noexcept;

template<class H2, class P2>
  void merge(unordered_map<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_map<Key, T, H2, P2, Allocator>&& source);
template<class H2, class P2>
  void merge(unordered_multimap<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_multimap<Key, T, H2, P2, Allocator>&& source);

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

```
  // map operations
  iterator         find(const key_type& k);
  const_iterator   find(const key_type& k) const;
  template <class K>
    iterator       find(const K& k);
  template <class K>
    const_iterator find(const K& k) const;
  size_type        count(const key_type& k) const;
  template <class K>
    size_type      count(const K& k) const;
  bool             contains(const key_type& k) const;
  template <class K>
    bool           contains(const K& k) const;
  pair<iterator, iterator>               equal_range(const key_type& k);
  pair<const_iterator, const_iterator>   equal_range(const key_type& k) const;
  template <class K>
    pair<iterator, iterator>             equal_range(const K& k);
  template <class K>
    pair<const_iterator, const_iterator> equal_range(const K& k) const;

  // 21.5.4.3, element access
  mapped_type& operator[](const key_type& k);
  mapped_type& operator[](key_type&& k);
  mapped_type& at(const key_type& k);
  const mapped_type& at(const key_type& k) const;

  // bucket interface
  size_type bucket_count() const noexcept;
  size_type max_bucket_count() const noexcept;
  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;
  const_local_iterator cbegin(size_type n) const;
  const_local_iterator cend(size_type n) const;

  // hash policy
  float load_factor() const noexcept;
  float max_load_factor() const noexcept;
  void max_load_factor(float z);
  void rehash(size_type n);
  void reserve(size_type n);
};

template<class InputIterator,
         class Hash = hash<iter-key-type<InputIterator>>,
         class Pred = equal_to<iter-key-type<InputIterator>>,
         class Allocator = allocator<iter-to-alloc-type<InputIterator>>>
  unordered_map(InputIterator, InputIterator, typename see below::size_type = see below,
                Hash = Hash(), Pred = Pred(), Allocator = Allocator())
    -> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash, Pred,
                     Allocator>;

template<class Key, class T, class Hash = hash<Key>,
         class Pred = equal_to<Key>, class Allocator = allocator<pair<const Key, T>>>
  unordered_map(initializer_list<pair<Key, T>>,
                typename see below::size_type = see below, Hash = Hash(),
                Pred = Pred(), Allocator = Allocator())
    -> unordered_map<Key, T, Hash, Pred, Allocator>;
```

```
  template<class InputIterator, class Allocator>
    unordered_map(InputIterator, InputIterator, typename see below::size_type, Allocator)
      -> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                       hash<iter-key-type<InputIterator>>,
                       equal_to<iter-key-type<InputIterator>>, Allocator>;

  template<class InputIterator, class Allocator>
    unordered_map(InputIterator, InputIterator, Allocator)
      -> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                       hash<iter-key-type<InputIterator>>,
                       equal_to<iter-key-type<InputIterator>>, Allocator>;

  template<class InputIterator, class Hash, class Allocator>
    unordered_map(InputIterator, InputIterator, typename see below::size_type, Hash, Allocator)
      -> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
                       equal_to<iter-key-type<InputIterator>>, Allocator>;

  template<class Key, class T, class Allocator>
    unordered_map(initializer_list<pair<Key, T>>, typename see below::size_type,
                  Allocator)
      -> unordered_map<Key, T, hash<Key>, equal_to<Key>, Allocator>;

  template<class Key, class T, class Allocator>
    unordered_map(initializer_list<pair<Key, T>>, Allocator)
      -> unordered_map<Key, T, hash<Key>, equal_to<Key>, Allocator>;

  template<class Key, class T, class Hash, class Allocator>
    unordered_map(initializer_list<pair<Key, T>>, typename see below::size_type, Hash,
                  Allocator)
      -> unordered_map<Key, T, Hash, equal_to<Key>, Allocator>;

  // swap
  template<class Key, class 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)
      noexcept(noexcept(x.swap(y)));
}
```

4 A `size_type` parameter type in an `unordered_map` deduction guide refers to the `size_type` member type of the type deduced by the deduction guide.

21.5.4.2 Constructors [unord.map.cnstr]

```
unordered_map() : unordered_map(size_type(see below)) { }
explicit unordered_map(size_type n,
                       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 predicate, and allocator, and using at least `n` buckets. For the default constructor, the number of buckets is implementation-defined. `max_load_factor()` returns `1.0`.

2 *Complexity:* Constant.

```
template<class InputIterator>
  unordered_map(InputIterator f, InputIterator l,
                size_type n = see below,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
unordered_map(initializer_list<value_type> il,
              size_type n = see below,
              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 predicate, 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)` for the first form, or from the range `[il.begin(), il.end())` for the second form. `max_load_factor()` returns `1.0`.

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

21.5.4.3 Element access [unord.map.elem]

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

1 *Effects:* Equivalent to: `return try_emplace(k).first->second;`

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

2 *Effects:* Equivalent to: `return try_emplace(move(k)).first->second;`

```
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.

21.5.4.4 Modifiers [unord.map.modifiers]

```
template<class P>
  pair<iterator, bool> insert(P&& obj);
```

1 *Constraints:* `is_constructible_v<value_type, P&&>` is `true`.

2 *Effects:* Equivalent to: `return emplace(std::forward<P>(obj));`

3 ~~*Remarks:* This signature shall not participate in overload resolution unless `is_constructible_v<value_type, P&&>` is `true`.~~

```
template<class P>
  iterator insert(const_iterator hint, P&& obj);
```

4 *Constraints:* `is_constructible_v<value_type, P&&>` is `true`.

5 *Effects:* Equivalent to: `return emplace_hint(hint, std::forward<P>(obj));`

6 ~~*Remarks:* This signature shall not participate in overload resolution unless `is_constructible_v<value_type, P&&>` is `true`.~~

```
template<class... Args>
  pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
```

7 ~~*Requires:*~~ *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `unordered_map` from `piecewise_construct`, `forward_as_tuple(k)`, `forward_as_tuple(std::forward<Args>(args)...)`.

8 *Effects:* If the map already contains an element whose key is equivalent to `k`, there is no effect. Otherwise inserts an object of type `value_type` constructed with `piecewise_construct`, `forward_as_tuple(k)`, `forward_as_tuple(std::forward<Args>(args)...)`.

9 *Returns:* In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to `k`.

10 *Complexity:* The same as `emplace` and `emplace_hint`, respectively.

```
template<class... Args>
  pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
```

11 ~~*Requires:*~~ *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `unordered_map` from `piecewise_construct`, `forward_as_tuple(std::move(k))`, `forward_as_tuple(std::forward<Args>(args)...)`.

12 *Effects:* If the map already contains an element whose key is equivalent to `k`, there is no effect. Otherwise inserts an object of type `value_type` constructed with `piecewise_construct`, `forward_as_tuple(std::move(k))`, `forward_as_tuple(std::forward<Args>(args)...)`.

13 *Returns:* In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to `k`.

14 *Complexity:* The same as `emplace` and `emplace_hint`, respectively.

```
template<class M>
  pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
  iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
```

15 ~~*Requires:*~~ *Mandates:* `is_assignable_v<mapped_type&, M&&>` shall be `true`.

16 *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `unordered_map` from `k`, `std::forward<M>(obj)`.

17 *Effects:* If the map already contains an element `e` whose key is equivalent to `k`, assigns `std::forward<M>(obj)` to `e.second`. Otherwise inserts an object of type `value_type` constructed with `k`, `std::forward<M>(obj)`.

18 *Returns:* In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to `k`.

19 *Complexity:* The same as `emplace` and `emplace_hint`, respectively.

```
template<class M>
  pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class M>
  iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);
```

20 ~~*Requires:*~~ *Mandates:* `is_assignable_v<mapped_type&, M&&>` shall be `true`.

21 *Expects:* `value_type` shall be *Cpp17EmplaceConstructible* into `unordered_map` from `std::move(k)`, `std::forward<M>(obj)`.

22 *Effects:* If the map already contains an element `e` whose key is equivalent to `k`, assigns `std::forward<M>(obj)` to `e.second`. Otherwise inserts an object of type `value_type` constructed with `std::move(k)`, `std::forward<M>(obj)`.

23 *Returns:* In the first overload, the `bool` component of the returned pair is `true` if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to `k`.

24 *Complexity:* The same as `emplace` and `emplace_hint`, respectively.

21.5.4.5 Erasure [unord.map.erasure]

```
template <class K, class T, class H, class P, class A, class Predicate>
  void erase_if(unordered_map<K, T, H, P, A>& c, Predicate pred);
```

1 *Effects:* Equivalent to:

```
for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
  } else {
    ++i;
  }
}
```

21.5.5 Class template `unordered_multimap` [unord.multimap]

21.5.5.1 Overview [unord.multimap.overview]

1 An `unordered_multimap` is an unordered associative container that supports equivalent keys (an instance of `unordered_multimap` may contain multiple copies of each key value) and that associates values of another type `mapped_type` with the keys. The `unordered_multimap` class supports forward iterators.

2 An `unordered_multimap` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 65). 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 subclause 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<class Key,
           class T,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Allocator = allocator<pair<const Key, T>>>
  class unordered_multimap {
  public:
    // types
    using key_type             = Key;
    using mapped_type          = T;
    using value_type           = pair<const Key, T>;
    using hasher               = Hash;
    using key_equal            = see 21.2.7;
    using allocator_type       = Allocator;
    using pointer              = typename allocator_traits<Allocator>::pointer;
    using const_pointer        = typename allocator_traits<Allocator>::const_pointer;
    using reference            = value_type&;
    using const_reference      = const value_type&;
    using size_type            = implementation-defined; // see 21.2
    using difference_type      = implementation-defined; // see 21.2

    using iterator             = implementation-defined; // see 21.2
    using const_iterator       = implementation-defined; // see 21.2
    using local_iterator       = implementation-defined; // see 21.2
    using const_local_iterator = implementation-defined; // see 21.2
    using node_type            = unspecified;

    // 21.5.5.2, construct/copy/destroy
    unordered_multimap();
    explicit unordered_multimap(size_type n,
                                const hasher& hf = hasher(),
                                const key_equal& eql = key_equal(),
                                const allocator_type& a = allocator_type());
    template<class InputIterator>
      unordered_multimap(InputIterator f, InputIterator l,
                         size_type n = see below,
                         const hasher& hf = hasher(),
                         const key_equal& eql = key_equal(),
                         const allocator_type& a = allocator_type());
    unordered_multimap(const unordered_multimap&);
    unordered_multimap(unordered_multimap&&);
    explicit unordered_multimap(const Allocator&);
    unordered_multimap(const unordered_multimap&, const Allocator&);
    unordered_multimap(unordered_multimap&&, const Allocator&);
    unordered_multimap(initializer_list<value_type> il,
                       size_type n = see below,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
    unordered_multimap(size_type n, const allocator_type& a)
      : unordered_multimap(n, hasher(), key_equal(), a) { }
    unordered_multimap(size_type n, const hasher& hf, const allocator_type& a)
      : unordered_multimap(n, hf, key_equal(), a) { }
    template<class InputIterator>
      unordered_multimap(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
        : unordered_multimap(f, l, n, hasher(), key_equal(), a) { }
```

```
template<class InputIterator>
  unordered_multimap(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                     const allocator_type& a)
    : unordered_multimap(f, l, n, hf, key_equal(), a) { }
unordered_multimap(initializer_list<value_type> il, size_type n, const allocator_type& a)
  : unordered_multimap(il, n, hasher(), key_equal(), a) { }
unordered_multimap(initializer_list<value_type> il, size_type n, const hasher& hf,
                   const allocator_type& a)
  : unordered_multimap(il, n, hf, key_equal(), a) { }
~unordered_multimap();
unordered_multimap& operator=(const unordered_multimap&);
unordered_multimap& operator=(unordered_multimap&&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_move_assignable_v<Hash> &&
           is_nothrow_move_assignable_v<Pred>);
unordered_multimap& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

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

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// 21.5.5.3, modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(value_type&& obj);
template<class P> iterator insert(P&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template<class P> iterator insert(const_iterator hint, P&& obj);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

iterator  erase(iterator position);
iterator  erase(const_iterator position);
size_type erase(const key_type& k);
iterator  erase(const_iterator first, const_iterator last);
void      swap(unordered_multimap&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_swappable_v<Hash> &&
           is_nothrow_swappable_v<Pred>);
void      clear() noexcept;

template<class H2, class P2>
  void merge(unordered_multimap<Key, T, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_multimap<Key, T, H2, P2, Allocator>&& source);
template<class H2, class P2>
  void merge(unordered_map<Key, T, H2, P2, Allocator>& source);
```

```
  template<class H2, class P2>
    void merge(unordered_map<Key, T, H2, P2, Allocator>&& source);

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

  // map operations
  iterator         find(const key_type& k);
  const_iterator   find(const key_type& k) const;
  template <class K>
    iterator       find(const K& k);
  template <class K>
    const_iterator find(const K& k) const;
  size_type        count(const key_type& k) const;
  template <class K>
    size_type      count(const K& k) const;
  bool             contains(const key_type& k) const;
  template <class K>
    bool           contains(const K& k) const;
  pair<iterator, iterator>               equal_range(const key_type& k);
  pair<const_iterator, const_iterator>   equal_range(const key_type& k) const;
  template <class K>
    pair<iterator, iterator>             equal_range(const K& k);
  template <class K>
    pair<const_iterator, const_iterator> equal_range(const K& k) const;

  // bucket interface
  size_type bucket_count() const noexcept;
  size_type max_bucket_count() const noexcept;
  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;
  const_local_iterator cbegin(size_type n) const;
  const_local_iterator cend(size_type n) const;

  // hash policy
  float load_factor() const noexcept;
  float max_load_factor() const noexcept;
  void max_load_factor(float z);
  void rehash(size_type n);
  void reserve(size_type n);
};

template<class InputIterator,
         class Hash = hash<iter-key-type<InputIterator>>,
         class Pred = equal_to<iter-key-type<InputIterator>>,
         class Allocator = allocator<iter-to-alloc-type<InputIterator>>>
  unordered_multimap(InputIterator, InputIterator,
                     typename see below::size_type = see below,
                     Hash = Hash(), Pred = Pred(), Allocator = Allocator())
    -> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                          Hash, Pred, Allocator>;

template<class Key, class T, class Hash = hash<Key>,
         class Pred = equal_to<Key>, class Allocator = allocator<pair<const Key, T>>>
  unordered_multimap(initializer_list<pair<Key, T>>,
                     typename see below::size_type = see below,
                     Hash = Hash(), Pred = Pred(), Allocator = Allocator())
    -> unordered_multimap<Key, T, Hash, Pred, Allocator>;
```

```
  template<class InputIterator, class Allocator>
    unordered_multimap(InputIterator, InputIterator, typename see below::size_type, Allocator)
      -> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                            hash<iter-key-type<InputIterator>>,
                            equal_to<iter-key-type<InputIterator>>, Allocator>;

  template<class InputIterator, class Allocator>
    unordered_multimap(InputIterator, InputIterator, Allocator)
      -> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
                            hash<iter-key-type<InputIterator>>,
                            equal_to<iter-key-type<InputIterator>>, Allocator>;

  template<class InputIterator, class Hash, class Allocator>
    unordered_multimap(InputIterator, InputIterator, typename see below::size_type, Hash,
                       Allocator)
      -> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
                            equal_to<iter-key-type<InputIterator>>, Allocator>;

  template<class Key, class T, class Allocator>
    unordered_multimap(initializer_list<pair<Key, T>>, typename see below::size_type,
                       Allocator)
      -> unordered_multimap<Key, T, hash<Key>, equal_to<Key>, Allocator>;

  template<class Key, class T, class Allocator>
    unordered_multimap(initializer_list<pair<Key, T>>, Allocator)
      -> unordered_multimap<Key, T, hash<Key>, equal_to<Key>, Allocator>;

  template<class Key, class T, class Hash, class Allocator>
    unordered_multimap(initializer_list<pair<Key, T>>, typename see below::size_type,
                       Hash, Allocator)
      -> unordered_multimap<Key, T, Hash, equal_to<Key>, Allocator>;

  // swap
  template<class Key, class 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)
      noexcept(noexcept(x.swap(y)));
}
```

4 A `size_type` parameter type in an `unordered_multimap` deduction guide refers to the `size_type` member type of the type deduced by the deduction guide.

21.5.5.2 Constructors [unord.multimap.cnstr]

```
unordered_multimap() : unordered_multimap(size_type(see below)) { }
explicit unordered_multimap(size_type n,
                            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 predicate, and allocator, and using at least `n` buckets. For the default constructor, the number of buckets is implementation-defined. `max_load_factor()` returns `1.0`.

2 *Complexity:* Constant.

```
template<class InputIterator>
  unordered_multimap(InputIterator f, InputIterator l,
                     size_type n = see below,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
unordered_multimap(initializer_list<value_type> il,
                   size_type n = see below,
                   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 predicate, 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)` for the first form, or from the range `[il.begin(), il.end())` for the second form. `max_load_factor()` returns `1.0`.

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

21.5.5.3 Modifiers [unord.multimap.modifiers]

```
template<class P>
  iterator insert(P&& obj);
```

1 *Constraints:* `is_constructible_v<value_type, P&&>` is `true`.

2 *Effects:* Equivalent to: `return emplace(std::forward<P>(obj));`

3 ~~*Remarks:* This signature shall not participate in overload resolution unless `is_constructible_v<value_type, P&&>` is `true`.~~

```
template<class P>
  iterator insert(const_iterator hint, P&& obj);
```

4 *Constraints:* `is_constructible_v<value_type, P&&>` is `true`.

5 *Effects:* Equivalent to: `return emplace_hint(hint, std::forward<P>(obj));`

6 ~~*Remarks:* This signature shall not participate in overload resolution unless `is_constructible_v<value_type, P&&>` is `true`.~~

21.5.5.4 Erasure [unord.multimap.erasure]

```
template <class K, class T, class H, class P, class A, class Predicate>
  void erase_if(unordered_multimap<K, T, H, P, A>& c, Predicate pred);
```

1 *Effects:* Equivalent to:

```
for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
  } else {
    ++i;
  }
}
```

21.5.6 Class template `unordered_set` [unord.set]

21.5.6.1 Overview [unord.set.overview]

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. The `unordered_set` class supports forward iterators.

2 An `unordered_set` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 65). 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<Key>` the `key type` and the value type are both `Key`. The `iterator` and `const_iterator` types are both constant iterator types. It is unspecified whether they are the same type.

3 This subclause 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 Key,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Allocator = allocator<Key>>
  class unordered_set {
  public:
    // types
    using key_type             = Key;
```

```
using value_type           = Key;
using hasher               = Hash;
using key_equal            = see 21.2.7;
using allocator_type       = Allocator;
using pointer              = typename allocator_traits<Allocator>::pointer;
using const_pointer        = typename allocator_traits<Allocator>::const_pointer;
using reference            = value_type&;
using const_reference      = const value_type&;
using size_type            = implementation-defined; // see 21.2
using difference_type      = implementation-defined; // see 21.2

using iterator             = implementation-defined; // see 21.2
using const_iterator       = implementation-defined; // see 21.2
using local_iterator       = implementation-defined; // see 21.2
using const_local_iterator = implementation-defined; // see 21.2
using node_type            = unspecified;
using insert_return_type   = insert-return-type<iterator, node_type>;

// 21.5.6.2, construct/copy/destroy
unordered_set();
explicit unordered_set(size_type n,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
template<class InputIterator>
  unordered_set(InputIterator f, InputIterator l,
                size_type n = see below,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
unordered_set(const unordered_set&);
unordered_set(unordered_set&&);
explicit unordered_set(const Allocator&);
unordered_set(const unordered_set&, const Allocator&);
unordered_set(unordered_set&&, const Allocator&);
unordered_set(initializer_list<value_type> il,
              size_type n = see below,
              const hasher& hf = hasher(),
              const key_equal& eql = key_equal(),
              const allocator_type& a = allocator_type());
unordered_set(size_type n, const allocator_type& a)
  : unordered_set(n, hasher(), key_equal(), a) { }
unordered_set(size_type n, const hasher& hf, const allocator_type& a)
  : unordered_set(n, hf, key_equal(), a) { }
template<class InputIterator>
  unordered_set(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
    : unordered_set(f, l, n, hasher(), key_equal(), a) { }
template<class InputIterator>
  unordered_set(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                const allocator_type& a)
  : unordered_set(f, l, n, hf, key_equal(), a) { }
unordered_set(initializer_list<value_type> il, size_type n, const allocator_type& a)
  : unordered_set(il, n, hasher(), key_equal(), a) { }
unordered_set(initializer_list<value_type> il, size_type n, const hasher& hf,
              const allocator_type& a)
  : unordered_set(il, n, hf, key_equal(), a) { }
~unordered_set();
unordered_set& operator=(const unordered_set&);
unordered_set& operator=(unordered_set&&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_move_assignable_v<Hash> &&
           is_nothrow_move_assignable_v<Pred>);
unordered_set& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;
```

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

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& obj);
pair<iterator, bool> insert(value_type&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
insert_return_type insert(node_type&& nh);
iterator           insert(const_iterator hint, node_type&& nh);

iterator  erase(iterator position);
iterator  erase(const_iterator position);
size_type erase(const key_type& k);
iterator  erase(const_iterator first, const_iterator last);
void      swap(unordered_set&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_swappable_v<Hash> &&
           is_nothrow_swappable_v<Pred>);
void      clear() noexcept;

template<class H2, class P2>
  void merge(unordered_set<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_set<Key, H2, P2, Allocator>&& source);
template<class H2, class P2>
  void merge(unordered_multiset<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_multiset<Key, H2, P2, Allocator>&& source);

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

// set operations
iterator         find(const key_type& k);
const_iterator   find(const key_type& k) const;
template <class K>
  iterator       find(const K& k);
template <class K>
  const_iterator find(const K& k) const;
size_type        count(const key_type& k) const;
template <class K>
  size_type      count(const K& k) const;
bool             contains(const key_type& k) const;
template <class K>
  bool           contains(const K& k) const;
```

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

  // bucket interface
  size_type bucket_count() const noexcept;
  size_type max_bucket_count() const noexcept;
  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;
  const_local_iterator cbegin(size_type n) const;
  const_local_iterator cend(size_type n) const;

  // hash policy
  float load_factor() const noexcept;
  float max_load_factor() const noexcept;
  void max_load_factor(float z);
  void rehash(size_type n);
  void reserve(size_type n);
};

template<class InputIterator,
         class Hash = hash<iter-value-type<InputIterator>>,
         class Pred = equal_to<iter-value-type<InputIterator>>,
         class Allocator = allocator<iter-value-type<InputIterator>>>
  unordered_set(InputIterator, InputIterator, typename see below::size_type = see below,
                Hash = Hash(), Pred = Pred(), Allocator = Allocator())
    -> unordered_set<iter-value-type<InputIterator>,
                     Hash, Pred, Allocator>;

template<class T, class Hash = hash<T>,
         class Pred = equal_to<T>, class Allocator = allocator<T>>
  unordered_set(initializer_list<T>, typename see below::size_type = see below,
                Hash = Hash(), Pred = Pred(), Allocator = Allocator())
    -> unordered_set<T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
  unordered_set(InputIterator, InputIterator, typename see below::size_type, Allocator)
    -> unordered_set<iter-value-type<InputIterator>,
                     hash<iter-value-type<InputIterator>>,
                     equal_to<iter-value-type<InputIterator>>,
                     Allocator>;

template<class InputIterator, class Hash, class Allocator>
  unordered_set(InputIterator, InputIterator, typename see below::size_type,
                Hash, Allocator)
    -> unordered_set<iter-value-type<InputIterator>, Hash,
                     equal_to<iter-value-type<InputIterator>>,
                     Allocator>;

template<class T, class Allocator>
  unordered_set(initializer_list<T>, typename see below::size_type, Allocator)
    -> unordered_set<T, hash<T>, equal_to<T>, Allocator>;

template<class T, class Hash, class Allocator>
  unordered_set(initializer_list<T>, typename see below::size_type, Hash, Allocator)
    -> unordered_set<T, Hash, equal_to<T>, Allocator>;
```

```
  // swap
  template<class Key, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
              unordered_set<Key, Hash, Pred, Alloc>& y)
      noexcept(noexcept(x.swap(y)));
}
```

4 A `size_type` parameter type in an `unordered_set` deduction guide refers to the `size_type` member type of the type deduced by the deduction guide.

21.5.6.2 Constructors [unord.set.cnstr]

```
unordered_set() : unordered_set(size_type(see below)) { }
explicit unordered_set(size_type n,
                       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 predicate, and allocator, and using at least `n` buckets. For the default constructor, the number of buckets is implementation-defined. `max_load_factor()` returns `1.0`.

2 *Complexity:* Constant.

```
template<class InputIterator>
  unordered_set(InputIterator f, InputIterator l,
                size_type n = see below,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
unordered_set(initializer_list<value_type> il,
              size_type n = see below,
              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 predicate, 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)` for the first form, or from the range `[il.begin(), il.end())` for the second form. `max_load_factor()` returns `1.0`.

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

21.5.6.3 Erasure [unord.set.erasure]

```
template <class K, class H, class P, class A, class Predicate>
  void erase_if(unordered_set<K, H, P, A>& c, Predicate pred);
```

1 *Effects:* Equivalent to:

```
for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
  } else {
    ++i;
  }
}
```

21.5.7 Class template `unordered_multiset` [unord.multiset]

21.5.7.1 Overview [unord.multiset.overview]

1 An `unordered_multiset` is an unordered associative container that supports equivalent keys (an instance of `unordered_multiset` may contain multiple copies of the same key value) and in which each element's key is the element itself. The `unordered_multiset` class supports forward iterators.

2 An `unordered_multiset` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 65). 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<Key>` the `key type` and the value type are both `Key`. The `iterator` and `const_iterator` types are both constant iterator types. It is unspecified whether they are the same type.

3 This subclause 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 Key,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Allocator = allocator<Key>>
  class unordered_multiset {
  public:
    // types
    using key_type             = Key;
    using value_type           = Key;
    using hasher               = Hash;
    using key_equal            = see 21.2.7;
    using allocator_type       = Allocator;
    using pointer              = typename allocator_traits<Allocator>::pointer;
    using const_pointer        = typename allocator_traits<Allocator>::const_pointer;
    using reference            = value_type&;
    using const_reference      = const value_type&;
    using size_type            = implementation-defined; // see 21.2
    using difference_type      = implementation-defined; // see 21.2

    using iterator             = implementation-defined; // see 21.2
    using const_iterator       = implementation-defined; // see 21.2
    using local_iterator       = implementation-defined; // see 21.2
    using const_local_iterator = implementation-defined; // see 21.2
    using node_type            = unspecified;

    // 21.5.7.2, construct/copy/destroy
    unordered_multiset();
    explicit unordered_multiset(size_type n,
                                const hasher& hf = hasher(),
                                const key_equal& eql = key_equal(),
                                const allocator_type& a = allocator_type());
    template<class InputIterator>
      unordered_multiset(InputIterator f, InputIterator l,
                         size_type n = see below,
                         const hasher& hf = hasher(),
                         const key_equal& eql = key_equal(),
                         const allocator_type& a = allocator_type());
    unordered_multiset(const unordered_multiset&);
    unordered_multiset(unordered_multiset&&);
    explicit unordered_multiset(const Allocator&);
    unordered_multiset(const unordered_multiset&, const Allocator&);
    unordered_multiset(unordered_multiset&&, const Allocator&);
    unordered_multiset(initializer_list<value_type> il,
                       size_type n = see below,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
    unordered_multiset(size_type n, const allocator_type& a)
      : unordered_multiset(n, hasher(), key_equal(), a) { }
    unordered_multiset(size_type n, const hasher& hf, const allocator_type& a)
      : unordered_multiset(n, hf, key_equal(), a) { }
    template<class InputIterator>
      unordered_multiset(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
        : unordered_multiset(f, l, n, hasher(), key_equal(), a) { }
    template<class InputIterator>
      unordered_multiset(InputIterator f, InputIterator l, size_type n, const hasher& hf,
                         const allocator_type& a)
```

```
  : unordered_multiset(f, l, n, hf, key_equal(), a) { }
unordered_multiset(initializer_list<value_type> il, size_type n, const allocator_type& a)
  : unordered_multiset(il, n, hasher(), key_equal(), a) { }
unordered_multiset(initializer_list<value_type> il, size_type n, const hasher& hf,
                   const allocator_type& a)
  : unordered_multiset(il, n, hf, key_equal(), a) { }
~unordered_multiset();
unordered_multiset& operator=(const unordered_multiset&);
unordered_multiset& operator=(unordered_multiset&&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_move_assignable_v<Hash> &&
           is_nothrow_move_assignable_v<Pred>);
unordered_multiset& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

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

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(value_type&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

iterator  erase(iterator position);
iterator  erase(const_iterator position);
size_type erase(const key_type& k);
iterator  erase(const_iterator first, const_iterator last);
void      swap(unordered_multiset&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
           is_nothrow_swappable_v<Hash> &&
           is_nothrow_swappable_v<Pred>);
void      clear() noexcept;

template<class H2, class P2>
  void merge(unordered_multiset<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_multiset<Key, H2, P2, Allocator>&& source);
template<class H2, class P2>
  void merge(unordered_set<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
  void merge(unordered_set<Key, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
```

```
  key_equal key_eq() const;

  // set operations
  iterator         find(const key_type& k);
  const_iterator   find(const key_type& k) const;
  template <class K>
    iterator       find(const K& k);
  template <class K>
    const_iterator find(const K& k) const;
  size_type        count(const key_type& k) const;
  template <class K>
    size_type      count(const K& k) const;
  bool             contains(const key_type& k) const;
  template <class K>
    bool           contains(const K& k) const;
  pair<iterator, iterator>               equal_range(const key_type& k);
  pair<const_iterator, const_iterator>   equal_range(const key_type& k) const;
  template <class K>
    pair<iterator, iterator>             equal_range(const K& k);
  template <class K>
    pair<const_iterator, const_iterator> equal_range(const K& k) const;

  // bucket interface
  size_type bucket_count() const noexcept;
  size_type max_bucket_count() const noexcept;
  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;
  const_local_iterator cbegin(size_type n) const;
  const_local_iterator cend(size_type n) const;

  // hash policy
  float load_factor() const noexcept;
  float max_load_factor() const noexcept;
  void max_load_factor(float z);
  void rehash(size_type n);
  void reserve(size_type n);
};

template<class InputIterator,
         class Hash = hash<iter-value-type<InputIterator>>,
         class Pred = equal_to<iter-value-type<InputIterator>>,
         class Allocator = allocator<iter-value-type<InputIterator>>>
  unordered_multiset(InputIterator, InputIterator, see below::size_type = see below,
                     Hash = Hash(), Pred = Pred(), Allocator = Allocator())
    -> unordered_multiset<iter-value-type<InputIterator>,
                          Hash, Pred, Allocator>;

template<class T, class Hash = hash<T>,
         class Pred = equal_to<T>, class Allocator = allocator<T>>
  unordered_multiset(initializer_list<T>, typename see below::size_type = see below,
                     Hash = Hash(), Pred = Pred(), Allocator = Allocator())
    -> unordered_multiset<T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
  unordered_multiset(InputIterator, InputIterator, typename see below::size_type, Allocator)
    -> unordered_multiset<iter-value-type<InputIterator>,
                          hash<iter-value-type<InputIterator>>,
                          equal_to<iter-value-type<InputIterator>>,
                          Allocator>;
```

```
  template<class InputIterator, class Hash, class Allocator>
    unordered_multiset(InputIterator, InputIterator, typename see below::size_type,
                       Hash, Allocator)
      -> unordered_multiset<iter-value-type<InputIterator>, Hash,
                            equal_to<iter-value-type<InputIterator>>,
                            Allocator>;

  template<class T, class Allocator>
    unordered_multiset(initializer_list<T>, typename see below::size_type, Allocator)
      -> unordered_multiset<T, hash<T>, equal_to<T>, Allocator>;

  template<class T, class Hash, class Allocator>
    unordered_multiset(initializer_list<T>, typename see below::size_type, Hash, Allocator)
      -> unordered_multiset<T, Hash, equal_to<T>, Allocator>;

  // swap
  template<class Key, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
              unordered_multiset<Key, Hash, Pred, Alloc>& y)
      noexcept(noexcept(x.swap(y)));
}
```

4 A `size_type` parameter type in an `unordered_multiset` deduction guide refers to the `size_type` member type of the type deduced by the deduction guide.

21.5.7.2 Constructors [unord.multiset.cnstr]

```
unordered_multiset() : unordered_multiset(size_type(see below)) { }
explicit unordered_multiset(size_type n,
                            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 predicate, and allocator, and using at least `n` buckets. For the default constructor, the number of buckets is implementation-defined. `max_load_factor()` returns `1.0`.

2 *Complexity:* Constant.

```
template<class InputIterator>
  unordered_multiset(InputIterator f, InputIterator l,
                     size_type n = see below,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
unordered_multiset(initializer_list<value_type> il,
                   size_type n = see below,
                   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 predicate, 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)` for the first form, or from the range `[il.begin(), il.end())` for the second form. `max_load_factor()` returns `1.0`.

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

21.5.7.3 Erasure [unord.multiset.erasure]

```
template <class K, class H, class P, class A, class Predicate>
  void erase_if(unordered_multiset<K, H, P, A>& c, Predicate pred);
```

1 *Effects:* Equivalent to:

```
for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
```

```
    } else {
      ++i;
    }
  }
```

21.6 Container adaptors [container.adaptors]

21.6.1 In general [container.adaptors.general]

1 The headers <queue> and <stack> define the container adaptors queue, priority_queue, and stack.

2 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. The first template parameter T of the container adaptors shall denote the same type as Container::value_type.

3 For container adaptors, no swap function throws an exception unless that exception is thrown by the swap of the adaptor's Container or Compare object (if any).

4 A deduction guide for a container adaptor shall not participate in overload resolution if any of the following are true:

(4.1) — It has an InputIterator template parameter and a type that does not qualify as an input iterator is deduced for that parameter.

(4.2) — It has a Compare template parameter and a type that qualifies as an allocator is deduced for that parameter.

(4.3) — It has a Container template parameter and a type that qualifies as an allocator is deduced for that parameter.

(4.4) — It has an Allocator template parameter and a type that does not qualify as an allocator is deduced for that parameter.

(4.5) — It has both Container and Allocator template parameters, and uses_allocator_v<Container, Allocator> is false.

21.6.2 Header <queue> synopsis [queue.syn]

```
#include <initializer_list>

namespace std {
  template<class T, class Container = deque<T>> class queue;
  template<class T, class Container = vector<T>,
           class Compare = less<typename Container::value_type>>
    class priority_queue;

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

  template<class T, class Container>
    void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));
  template<class T, class Container, class Compare>
    void swap(priority_queue<T, Container, Compare>& x,
              priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));
}
```

21.6.3 Header <stack> synopsis [stack.syn]

```
#include <initializer_list>

namespace std {
  template<class T, class Container = deque<T>> class stack;

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

  template<class T, class Container>
    void swap(stack<T, Container>& x, stack<T, Container>& y) noexcept(noexcept(x.swap(y)));
}
```

21.6.4 Class template queue [queue]

21.6.4.1 Definition [queue.defn]

1 Any sequence container supporting operations `front()`, `back()`, `push_back()` and `pop_front()` can be used to instantiate `queue`. In particular, `list` (21.3.10) and `deque` (21.3.8) can be used.

```
namespace std {
  template<class T, class Container = deque<T>>
  class queue {
  public:
    using value_type      = typename Container::value_type;
    using reference       = typename Container::reference;
    using const_reference = typename Container::const_reference;
    using size_type       = typename Container::size_type;
    using container_type  =          Container;

  protected:
    Container c;

  public:
    queue() : queue(Container()) {}
    explicit queue(const Container&);
    explicit queue(Container&&);
    template<class Alloc> explicit queue(const Alloc&);
    template<class Alloc> queue(const Container&, const Alloc&);
    template<class Alloc> queue(Container&&, const Alloc&);
    template<class Alloc> queue(const queue&, const Alloc&);
    template<class Alloc> queue(queue&&, const Alloc&);

    [[nodiscard]] bool empty() const    { return c.empty(); }
    size_type         size()  const     { return c.size(); }
    reference         front()           { return c.front(); }
    const_reference   front() const     { return c.front(); }
    reference         back()            { return c.back(); }
    const_reference   back() const      { return c.back(); }
    void push(const value_type& x)      { c.push_back(x); }
    void push(value_type&& x)           { c.push_back(std::move(x)); }
    template<class... Args>
      decltype(auto) emplace(Args&&... args)
        { return c.emplace_back(std::forward<Args>(args)...); }
    void pop()                          { c.pop_front(); }
```

```
    void swap(queue& q) noexcept(is_nothrow_swappable_v<Container>)
      { using std::swap; swap(c, q.c); }
  };

  template<class Container>
    queue(Container) -> queue<typename Container::value_type, Container>;

  template<class Container, class Allocator>
    queue(Container, Allocator) -> queue<typename Container::value_type, Container>;

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

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

21.6.4.2 Constructors [queue.cons]

```
explicit queue(const Container& cont);
```

1 *Effects:* Initializes `c` with `cont`.

```
explicit queue(Container&& cont);
```

2 *Effects:* Initializes `c` with `std::move(cont)`.

21.6.4.3 Constructors with allocators [queue.cons.alloc]

1 If `uses_allocator_v<container_type, Alloc>` is `false` the constructors in this subclause shall not participate in overload resolution.

```
template<class Alloc> explicit queue(const Alloc& a);
```

2 *Effects:* Initializes `c` with `a`.

```
template<class Alloc> queue(const container_type& cont, const Alloc& a);
```

3 *Effects:* Initializes `c` with `cont` as the first argument and `a` as the second argument.

```
template<class Alloc> queue(container_type&& cont, const Alloc& a);
```

4 *Effects:* Initializes `c` with `std::move(cont)` as the first argument and `a` as the second argument.

```
template<class Alloc> queue(const queue& q, const Alloc& a);
```

5 *Effects:* Initializes `c` with `q.c` as the first argument and `a` as the second argument.

```
template<class Alloc> queue(queue&& q, const Alloc& a);
```

6 *Effects:* Initializes `c` with `std::move(q.c)` as the first argument and `a` as the second argument.

21.6.4.4 Operators [queue.ops]

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

1 *Returns:* `x.c == y.c`.

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

2 *Returns:* `x.c != y.c`.

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

3 *Returns:* `x.c < y.c`.

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

4 *Returns:* `x.c > y.c`.

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

5 *Returns:* `x.c <= y.c`.

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

6 *Returns:* `x.c >= y.c`.

21.6.4.5 Specialized algorithms [queue.special]

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

1 ~~*Remarks:*~~ *Constraints:* ~~This function shall not participate in overload resolution unless~~ `is_swappable_v<Container>` is `true`.

2 *Effects:* As if by `x.swap(y)`.

21.6.5 Class template `priority_queue` [priority.queue]

21.6.5.1 Overview [priqueue.overview]

1 Any sequence container with random access iterator and supporting operations `front()`, `push_back()` and `pop_back()` can be used to instantiate `priority_queue`. In particular, `vector` (21.3.11) and `deque` (21.3.8) 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 T, class Container = vector<T>,
           class Compare = less<typename Container::value_type>>
  class priority_queue {
  public:
    using value_type      = typename Container::value_type;
    using reference       = typename Container::reference;
    using const_reference = typename Container::const_reference;
    using size_type       = typename Container::size_type;
    using container_type  = Container;
    using value_compare   = Compare;

  protected:
    Container c;
    Compare comp;

  public:
    priority_queue() : priority_queue(Compare()) {}
    explicit priority_queue(const Compare& x) : priority_queue(x, Container()) {}
    priority_queue(const Compare& x, const Container&);
    priority_queue(const Compare& x, Container&&);
    template<class InputIterator>
      priority_queue(InputIterator first, InputIterator last, const Compare& x,
                     const Container&);
    template<class InputIterator>
      priority_queue(InputIterator first, InputIterator last,
                     const Compare& x = Compare(), Container&& = Container());
    template<class Alloc> explicit priority_queue(const Alloc&);
    template<class Alloc> priority_queue(const Compare&, const Alloc&);
    template<class Alloc> priority_queue(const Compare&, const Container&, const Alloc&);
    template<class Alloc> priority_queue(const Compare&, Container&&, const Alloc&);
    template<class Alloc> priority_queue(const priority_queue&, const Alloc&);
    template<class Alloc> priority_queue(priority_queue&&, const Alloc&);
```

```
    [[nodiscard]] bool empty() const { return c.empty(); }
    size_type size()  const          { return c.size(); }
    const_reference   top() const    { return c.front(); }
    void push(const value_type& x);
    void push(value_type&& x);
    template<class... Args> void emplace(Args&&... args);
    void pop();
    void swap(priority_queue& q) noexcept(is_nothrow_swappable_v<Container> &&
                                          is_nothrow_swappable_v<Compare>)
      { using std::swap; swap(c, q.c); swap(comp, q.comp); }
  };

  template<class Compare, class Container>
    priority_queue(Compare, Container)
      -> priority_queue<typename Container::value_type, Container, Compare>;

  template<class InputIterator,
           class Compare = less<typename iterator_traits<InputIterator>::value_type>,
           class Container = vector<typename iterator_traits<InputIterator>::value_type>>
    priority_queue(InputIterator, InputIterator, Compare = Compare(), Container = Container())
      -> priority_queue<typename iterator_traits<InputIterator>::value_type, Container, Compare>;

  template<class Compare, class Container, class Allocator>
    priority_queue(Compare, Container, Allocator)
      -> priority_queue<typename Container::value_type, Container, Compare>;

  // no equality is provided

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

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

21.6.5.2 Constructors [priqueue.cons]

```
priority_queue(const Compare& x, const Container& y);
priority_queue(const Compare& x, Container&& y);
```

1 ~~*Requires:*~~ *Expects:* `x` 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.end(), comp)`.

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

3 ~~*Requires:*~~ *Expects:* `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 `c.insert(c.end(), first, last)`; and finally calls `make_heap(c.begin(), c.end(), comp)`.

21.6.5.3 Constructors with allocators [priqueue.cons.alloc]

1 If `uses_allocator_v<container_type, Alloc>` is `false` the constructors in this subclause shall not participate in overload resolution.

```
template<class Alloc> explicit priority_queue(const Alloc& a);
```

2 *Effects:* Initializes `c` with `a` and value-initializes `comp`.

```
template<class Alloc> priority_queue(const Compare& compare, const Alloc& a);
```

3 *Effects:* Initializes `c` with `a` and initializes `comp` with `compare`.

```
template<class Alloc>
  priority_queue(const Compare& compare, const Container& cont, const Alloc& a);
```

4 *Effects:* Initializes `c` with `cont` as the first argument and `a` as the second argument, and initializes `comp` with `compare`; calls `make_heap(c.begin(), c.end(), comp)`.

```
template<class Alloc>
  priority_queue(const Compare& compare, Container&& cont, const Alloc& a);
```

5 *Effects:* Initializes `c` with `std::move(cont)` as the first argument and `a` as the second argument, and initializes `comp` with `compare`; calls `make_heap(c.begin(), c.end(), comp)`.

```
template<class Alloc> priority_queue(const priority_queue& q, const Alloc& a);
```

6 *Effects:* Initializes `c` with `q.c` as the first argument and `a` as the second argument, and initializes `comp` with `q.comp`.

```
template<class Alloc> priority_queue(priority_queue&& q, const Alloc& a);
```

7 *Effects:* Initializes `c` with `std::move(q.c)` as the first argument and `a` as the second argument, and initializes `comp` with `std::move(q.comp)`.

21.6.5.4 Members [priqueue.members]

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

1 *Effects:* As if by:

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

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

2 *Effects:* As if by:

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

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

3 *Effects:* As if by:

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

```
void pop();
```

4 *Effects:* As if by:

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

21.6.5.5 Specialized algorithms [priqueue.special]

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

1 ~~*Remarks:*~~ *Constraints:* ~~This function shall not participate in overload resolution unless~~ `is_swappable_v<Container>` is `true` and `is_swappable_v<Compare>` is `true`.

2 *Effects:* As if by `x.swap(y)`.

21.6.6 Class template `stack` [stack]

1 Any sequence container supporting operations `back()`, `push_back()` and `pop_back()` can be used to instantiate `stack`. In particular, `vector` (21.3.11), `list` (21.3.10) and `deque` (21.3.8) can be used.

21.6.6.1 Definition [stack.defn]

```
namespace std {
  template<class T, class Container = deque<T>>
  class stack {
  public:
    using value_type      = typename Container::value_type;
    using reference       = typename Container::reference;
    using const_reference = typename Container::const_reference;
    using size_type       = typename Container::size_type;
    using container_type  = Container;

  protected:
    Container c;

  public:
    stack() : stack(Container()) {}
    explicit stack(const Container&);
    explicit stack(Container&&);
    template<class Alloc> explicit stack(const Alloc&);
    template<class Alloc> stack(const Container&, const Alloc&);
    template<class Alloc> stack(Container&&, const Alloc&);
    template<class Alloc> stack(const stack&, const Alloc&);
    template<class Alloc> stack(stack&&, const Alloc&);

    [[nodiscard]] bool empty() const    { return c.empty(); }
    size_type size()  const             { return c.size(); }
    reference         top()             { return c.back(); }
    const_reference   top() const       { return c.back(); }
    void push(const value_type& x)      { c.push_back(x); }
    void push(value_type&& x)           { c.push_back(std::move(x)); }
    template<class... Args>
      decltype(auto) emplace(Args&&... args)
        { return c.emplace_back(std::forward<Args>(args)...); }
    void pop()                          { c.pop_back(); }
    void swap(stack& s) noexcept(is_nothrow_swappable_v<Container>)
      { using std::swap; swap(c, s.c); }
  };

  template<class Container>
    stack(Container) -> stack<typename Container::value_type, Container>;

  template<class Container, class Allocator>
    stack(Container, Allocator) -> stack<typename Container::value_type, Container>;

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

21.6.6.2 Constructors [stack.cons]

```
explicit stack(const Container& cont);
```

1 *Effects:* Initializes `c` with `cont`.

```
explicit stack(Container&& cont);
```

2 *Effects:* Initializes `c` with `std::move(cont)`.

21.6.6.3 Constructors with allocators [stack.cons.alloc]

1 If `uses_allocator_v<container_type, Alloc>` is `false` the constructors in this subclause shall not participate in overload resolution.

```
template<class Alloc> explicit stack(const Alloc& a);
```

2 *Effects:* Initializes `c` with `a`.

```
template<class Alloc> stack(const container_type& cont, const Alloc& a);
```

3 *Effects:* Initializes `c` with `cont` as the first argument and `a` as the second argument.

```
template<class Alloc> stack(container_type&& cont, const Alloc& a);
```

4 *Effects:* Initializes `c` with `std::move(cont)` as the first argument and `a` as the second argument.

```
template<class Alloc> stack(const stack& s, const Alloc& a);
```

5 *Effects:* Initializes `c` with `s.c` as the first argument and `a` as the second argument.

```
template<class Alloc> stack(stack&& s, const Alloc& a);
```

6 *Effects:* Initializes `c` with `std::move(s.c)` as the first argument and `a` as the second argument.

21.6.6.4 Operators [stack.ops]

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

1 *Returns:* `x.c == y.c`.

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

2 *Returns:* `x.c != y.c`.

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

3 *Returns:* `x.c < y.c`.

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

4 *Returns:* `x.c > y.c`.

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

5 *Returns:* `x.c <= y.c`.

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

6 *Returns:* `x.c >= y.c`.

21.6.6.5 Specialized algorithms [stack.special]

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

1 ~~*Remarks:*~~ *Constraints:* ~~This function shall not participate in overload resolution unless~~ `is_swappable_v<Container>` is `true`.

2 *Effects:* As if by `x.swap(y)`.

21.7 Views [views]

21.7.1 General [views.general]

1 The header `<span>` defines the view `span`.

21.7.2 Header synopsis [span.syn]

```
namespace std {
  // constants
  inline constexpr ptrdiff_t dynamic_extent = -1;

  // 21.7.3, class template span
  template<class ElementType, ptrdiff_t Extent = dynamic_extent>
    class span;
```

```
  // 21.7.3.7, views of object representation
  template<class ElementType, ptrdiff_t Extent>
    span<const byte,
         Extent == dynamic_extent ? dynamic_extent
                                  : static_cast<ptrdiff_t>(sizeof(ElementType)) * Extent>
      as_bytes(span<ElementType, Extent> s) noexcept;

  template<class ElementType, ptrdiff_t Extent>
    span<byte,
         Extent == dynamic_extent ? dynamic_extent
                                  : static_cast<ptrdiff_t>(sizeof(ElementType)) * Extent>
      as_writable_bytes(span<ElementType, Extent> s) noexcept;
}
```

21.7.3 Class template span [views.span]

21.7.3.1 Overview [span.overview]

1 A `span` is a view over a contiguous sequence of objects, the storage of which is owned by some other object.

2 The iterator types `span::iterator` and `span::const_iterator` model `ContiguousIterator` (**??**), meet the *Cpp17RandomAccessIterator* requirements (**??**), and meet the requirements for constexpr iterators (**??**). All requirements on container iterators (21.2) apply to `span::iterator` and `span::const_iterator` as well.

3 All member functions of `span` have constant time complexity.

```
namespace std {
  template<class ElementType, ptrdiff_t Extent = dynamic_extent>
  class span {
  public:
    // constants and types
    using element_type = ElementType;
    using value_type = remove_cv_t<ElementType>;
    using index_type = ptrdiff_t;
    using difference_type = ptrdiff_t;
    using pointer = element_type*;
    using reference = element_type&;
    using iterator = implementation-defined;
    using const_iterator = implementation-defined;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    static constexpr index_type extent = Extent;

    // 21.7.3.2, constructors, copy, and assignment
    constexpr span() noexcept;
    constexpr span(pointer ptr, index_type count);
    constexpr span(pointer first, pointer last);
    template<size_t N>
      constexpr span(element_type (&arr)[N]) noexcept;
    template<size_t N>
      constexpr span(array<value_type, N>& arr) noexcept;
    template<size_t N>
      constexpr span(const array<value_type, N>& arr) noexcept;
    template<class Container>
      constexpr span(Container& cont);
    template<class Container>
      constexpr span(const Container& cont);
    constexpr span(const span& other) noexcept = default;
    template<class OtherElementType, ptrdiff_t OtherExtent>
      constexpr span(const span<OtherElementType, OtherExtent>& s) noexcept;

    ~span() noexcept = default;

    constexpr span& operator=(const span& other) noexcept = default;
```

```
    // 21.7.3.3, subviews
    template<ptrdiff_t Count>
      constexpr span<element_type, Count> first() const;
    template<ptrdiff_t Count>
      constexpr span<element_type, Count> last() const;
    template<ptrdiff_t Offset, ptrdiff_t Count = dynamic_extent>
      constexpr span<element_type, see below> subspan() const;

    constexpr span<element_type, dynamic_extent> first(index_type count) const;
    constexpr span<element_type, dynamic_extent> last(index_type count) const;
    constexpr span<element_type, dynamic_extent> subspan(
      index_type offset, index_type count = dynamic_extent) const;

    // 21.7.3.4, observers
    constexpr index_type size() const noexcept;
    constexpr index_type size_bytes() const noexcept;
    constexpr bool empty() const noexcept;

    // 21.7.3.5, element access
    constexpr reference operator[](index_type idx) const;
    constexpr reference operator()(index_type idx) const;
    constexpr pointer data() const noexcept;

    // 21.7.3.6, iterator support
    constexpr iterator begin() const noexcept;
    constexpr iterator end() const noexcept;
    constexpr const_iterator cbegin() const noexcept;
    constexpr const_iterator cend() const noexcept;
    constexpr reverse_iterator rbegin() const noexcept;
    constexpr reverse_iterator rend() const noexcept;
    constexpr const_reverse_iterator crbegin() const noexcept;
    constexpr const_reverse_iterator crend() const noexcept;

    friend constexpr iterator begin(span s) noexcept { return s.begin(); }
    friend constexpr iterator end(span s) noexcept { return s.end(); }

  private:
    pointer data_;     // exposition only
    index_type size_; // exposition only
  };

  template<class T, size_t N>
    span(T (&)[N]) -> span<T, N>;
  template<class T, size_t N>
    span(array<T, N>&) -> span<T, N>;
  template<class T, size_t N>
    span(const array<T, N>&) -> span<const T, N>;
  template<class Container>
    span(Container&) -> span<typename Container::value_type>;
  template<class Container>
    span(const Container&) -> span<const typename Container::value_type>;
}
```

4 `ElementType` is required to be a complete object type that is not an abstract class type.

5 If `Extent` is negative and not equal to `dynamic_extent`, the program is ill-formed.

21.7.3.2 Constructors, copy, and assignment [span.cons]

```
constexpr span() noexcept;
```

1 *Constraints:* `Extent <= 0` is `true`.

2 *Ensures:* `size() == 0 && data() == nullptr`.

3 ~~*Remarks:* This constructor shall not participate in overload resolution unless `Extent <= 0` is `true`.~~

```
constexpr span(pointer ptr, index_type count);
```

4 ~~*Requires:*~~ *Expects:* `[ptr, ptr + count)` shall be a valid range. If `extent` is not equal to `dynamic_extent`, then `count` shall be equal to `extent`.

5 *Effects:* Constructs a `span` that is a view over the range `[ptr, ptr + count)`.

6 *Ensures:* `size() == count && data() == ptr`.

7 *Throws:* Nothing.

```
constexpr span(pointer first, pointer last);
```

8 ~~*Requires:*~~ *Expects:* `[first, last)` shall be a valid range. If `extent` is not equal to `dynamic_extent`, then `last - first` shall be equal to `extent`.

9 *Effects:* Constructs a span that is a view over the range `[first, last)`.

10 *Ensures:* `size() == last - first && data() == first`.

11 *Throws:* Nothing.

```
template<size_t N> constexpr span(element_type (&arr)[N]) noexcept;
template<size_t N> constexpr span(array<value_type, N>& arr) noexcept;
template<size_t N> constexpr span(const array<value_type, N>& arr) noexcept;
```

12 *Constraints:*

(12.1) — `extent == dynamic_extent || N == extent` is `true`, and

(12.2) — `remove_pointer_t<decltype(data(arr))>(*)[]` is convertible to `ElementType(*)[]`.

13 *Effects:* Constructs a `span` that is a view over the supplied array.

14 *Ensures:* `size() == N && data() == data(arr)`.

15 *Remarks:* These constructors shall not participate in overload resolution unless:

(15.1) — `extent == dynamic_extent || N == extent` is `true`, and

(15.2) — `remove_pointer_t<decltype(data(arr))>(*)[]` is convertible to `ElementType(*)[]`.

```
template<class Container> constexpr span(Container& cont);
template<class Container> constexpr span(const Container& cont);
```

16 ~~*Requires:*~~ *Expects:* `[data(cont), data(cont) + size(cont))` shall be a valid range. If `extent` is not equal to `dynamic_extent`, then `size(cont)` shall be equal to `extent`.

17 *Constraints:*

(17.1) — `Container` is not a specialization of `span`,

(17.2) — `Container` is not a specialization of `array`,

(17.3) — `is_array_v<Container>` is `false`,

(17.4) — `data(cont)` and `size(cont)` are both well-formed, and

(17.5) — `remove_pointer_t<decltype(data(cont))>(*)[]` is convertible to `ElementType(*)[]`.

18 *Effects:* Constructs a `span` that is a view over the range `[data(cont), data(cont) + size(cont))`.

19 *Ensures:* `size() == size(cont) && data() == data(cont)`.

20 *Throws:* What and when `data(cont)` and `size(cont)` throw.

21 *Remarks:* These constructors shall not participate in overload resolution unless:

(21.1) — `Container` is not a specialization of `span`,

(21.2) — `Container` is not a specialization of `array`,

(21.3) — `is_array_v<Container>` is `false`,

(21.4) — `data(cont)` and `size(cont)` are both well-formed, and

(21.5) — `remove_pointer_t<decltype(data(cont))>(*)[]` is convertible to `ElementType(*)[]`.

```
constexpr span(const span& other) noexcept = default;
```

22 *Ensures:* `other.size() == size() && other.data() == data()`.

```
template<class OtherElementType, ptrdiff_t OtherExtent>
  constexpr span(const span<OtherElementType, OtherExtent>& s) noexcept;
```

23 *Constraints:*

(23.1) — `Extent == dynamic_extent || Extent == OtherExtent` is `true`, and

(23.2) — `OtherElementType(*)[]` is convertible to `ElementType(*)[]`.

24 *Effects:* Constructs a `span` that is a view over the range `[s.data(), s.data() + s.size())`.

25 *Ensures:* `size() == s.size() && data() == s.data()`.

26 *Remarks:* This constructor shall not participate in overload resolution unless:

(26.1) — `Extent == dynamic_extent || Extent == OtherExtent` is `true`, and

(26.2) — `OtherElementType(*)[]` is convertible to `ElementType(*)[]`.

```
constexpr span& operator=(const span& other) noexcept = default;
```

27 *Ensures:* `size() == other.size() && data() == other.data()`.

21.7.3.3 Subviews [span.sub]

```
template<ptrdiff_t Count> constexpr span<element_type, Count> first() const;
```

1 ~~*Requires:*~~ *Expects:* `0 <= Count && Count <= size()`.

2 *Effects:* Equivalent to: `return {data(), Count};`

```
template<ptrdiff_t Count> constexpr span<element_type, Count> last() const;
```

3 ~~*Requires:*~~ *Expects:* `0 <= Count && Count <= size()`.

4 *Effects:* Equivalent to: `return {data() + (size() - Count), Count};`

```
template<ptrdiff_t Offset, ptrdiff_t Count = dynamic_extent>
  constexpr span<element_type, see below> subspan() const;
```

5 ~~*Requires:*~~ *Expects:*

```
(0 <= Offset && Offset <= size())
&& (Count == dynamic_extent || Count >= 0 && Offset + Count <= size())
```

6 *Effects:* Equivalent to:

```
return span<ElementType, see below>(
  data() + Offset, Count != dynamic_extent ? Count : size() - Offset);
```

7 *Remarks:* The second template argument of the returned `span` type is:

```
Count != dynamic_extent ? Count
                        : (Extent != dynamic_extent ? Extent - Offset
                                                    : dynamic_extent)
```

```
constexpr span<element_type, dynamic_extent> first(index_type count) const;
```

8 ~~*Requires:*~~ *Expects:* `0 <= count && count <= size()`.

9 *Effects:* Equivalent to: `return {data(), count};`

```
constexpr span<element_type, dynamic_extent> last(index_type count) const;
```

10 ~~*Requires:*~~ *Expects:* `0 <= count 0 && count <= size()`.

11 *Effects:* Equivalent to: `return {data() + (size() - count), count};`

```
constexpr span<element_type, dynamic_extent> subspan(
  index_type offset, index_type count = dynamic_extent) const;
```

12 ~~*Requires:*~~ *Expects:*

```
(0 <= offset && offset <= size())
&& (count == dynamic_extent || count >= 0 && offset + count <= size())
```

13 *Effects:* Equivalent to:

```
return {data() + offset, count == dynamic_extent ? size() - offset : count};
```

21.7.3.4 Observers [span.obs]

```
constexpr index_type size() const noexcept;
```

1 *Effects:* Equivalent to: `return size_;`

```
constexpr index_type size_bytes() const noexcept;
```

2 *Effects:* Equivalent to: `return size() * sizeof(element_type);`

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

3 *Effects:* Equivalent to: `return size() == 0;`

21.7.3.5 Element access [span.elem]

```
constexpr reference operator[](index_type idx) const;
constexpr reference operator()(index_type idx) const;
```

1 ~~*Requires:*~~ *Expects:* `0 <= idx && idx < size()`.

2 *Effects:* Equivalent to: `return *(data() + idx);`

```
constexpr pointer data() const noexcept;
```

3 *Effects:* Equivalent to: `return data_;`

21.7.3.6 Iterator support [span.iterators]

```
constexpr iterator begin() const noexcept;
```

1 *Returns:* An iterator referring to the first element in the span. If `empty()` is `true`, then it returns the same value as `end()`.

```
constexpr iterator end() const noexcept;
```

2 *Returns:* An iterator which is the past-the-end value.

```
constexpr reverse_iterator rbegin() const noexcept;
```

3 *Effects:* Equivalent to: `return reverse_iterator(end());`

```
constexpr reverse_iterator rend() const noexcept;
```

4 *Returns:* Equivalent to: `return reverse_iterator(begin());`

```
constexpr const_iterator cbegin() const noexcept;
```

5 *Returns:* A constant iterator referring to the first element in the span. If `empty()` is `true`, then it returns the same value as `cend()`.

```
constexpr const_iterator cend() const noexcept;
```

6 *Returns:* A constant iterator which is the past-the-end value.

```
constexpr const_reverse_iterator crbegin() const noexcept;
```

7 *Effects:* Equivalent to: `return const_reverse_iterator(cend());`

```
constexpr const_reverse_iterator crend() const noexcept;
```

8 *Effects:* Equivalent to: `return const_reverse_iterator(cbegin());`

21.7.3.7 Views of object representation [span.objectrep]

```
template<class ElementType, ptrdiff_t Extent>
  span<const byte,
       Extent == dynamic_extent ? dynamic_extent
                                : static_cast<ptrdiff_t>(sizeof(ElementType)) * Extent>
    as_bytes(span<ElementType, Extent> s) noexcept;
```

1 *Effects:* Equivalent to: `return {reinterpret_cast<const byte*>(s.data()), s.size_bytes()};`

```
template<class ElementType, ptrdiff_t Extent>
  span<byte,
       Extent == dynamic_extent ? dynamic_extent
                                : static_cast<ptrdiff_t>(sizeof(ElementType)) * Extent>
    as_writable_bytes(span<ElementType, Extent> s) noexcept;
```

2 *Constraints:* `is_const_v<ElementType>` is `false`.

3 *Effects:* Equivalent to: `return {reinterpret_cast<byte*>(s.data()), s.size_bytes()};`

4 ~~*Remarks:* This function shall not participate in overload resolution unless `is_const_v<ElementType>` is `false`.~~