Conversions from ranges to containers

Document #:	P1206R4
Date:	2021-06-14
Project:	Programming Language C++
Audience:	SG-9
Reply-to:	Corentin Jabot <corentin.jabot@gmail.com> Eric Niebler <eric.niebler@gmail.com> Casey Carter <casey@carter.net>

Abstract

We propose facilities to make constructing containers from ranges more convenient.

Revisions

Revision 4

- Add `from_range_t` and methods taking ranges to most containers
- Improve the wording of `ranges::to`
- `ranges::to` calls reserve when possible
- Rewrite the motivation

Revision 3

- Add support for `from_range_t`
- Add support for nested containers
- Remove syntax without parenthesis

Revision 2

- Remove the implicit const removal when converting an associative container to a container of pairs
- Use CTAD to determine the value type of the returned container
- Attempt at wording

Revision 1

- Split out the proposed constructors for string view and span into separate papers ([**?**] and [**?**] respectively)
- Use a function based approach rather than adding a constructor to standard containers, as it proved unworkable.

Overview

We propose 2 facilites to make it easier to construct container and from ranges:

- `ranges::to` a function that can materialize any range as a container, including non-standard containers, and recursive containers
- tagged constructors, `insert` and `assign` methods for standard containers and string types.

We propose that all the following syntaxes be valid constructs. The examples are meant to be illustrative of the interface's capabilities - the primary use case for it is to materialize views, even if copying from one container type to another is possible.

```
auto l = std::views::iota(1, 10);

// create a vector with the elements of l
auto vec = ranges::to<std::vector<int>>(l); // or vector{std::from_range, l};

//Specify an allocator
auto b = ranges::to<std::vector<int, Alloc>>(l, alloc); // or vector{std::from_range, l, alloc};

//deducing value_type
auto c = ranges::to<std::vector>(l);

// explicit conversion int -> long
auto d = ranges::to<std::vector<long>>(l);


//Supports converting associative container to sequence containers
auto f = ranges::to<vector>(m);

//Supports converting sequence containers to associative ones
auto g = ranges::to<map>(f);

//Pipe syntaxe
auto g = l | ranges::view::take(42) | ranges::to<std::vector>();

//Pipe syntax with allocator
auto h = l | ranges::view::take(42) | ranges::to<std::vector>(alloc);

//The pipe syntax also support specifying the type and conversions
auto i = l | ranges::view::take(42) | ranges::to<std::vector<long>>();
```

```cpp
// Nested ranges
std::list<std::forward_list<int>> lst = {{0, 1, 2, 3}, {4, 5, 6, 7}};
auto vec1 = ranges::to<std::vector<std::vector<int>>>(lst);
auto vec2 = ranges::to<std::vector<std::deque<double>>>(lst);
```

Tony tables

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><pre>std::list<int> lst = /*...*/;
std::vector<int> vec
    {std::begin(lst), std::end(lst)};</pre></td><td><pre>std::vector<int> vec = lst | ranges::to<std::vector>();</pre></td></tr>
<tr><td><pre>auto view = ranges::iota(42);
vector <
  iter_value_t<
    iterator_t<decltype(view)>
  >
> vec;
if constexpr(SizedRanged<decltype(view)>) {
  vec.reserve(ranges::size(view)));
}
ranges::copy(view, std::back_inserter(vec));</pre></td><td><pre>auto vec = ranges::iota(0, 42)
    | ranges::to<std::vector>();</pre></td></tr>
<tr><td><pre>std::map<int, widget> map = get_widgets_map();
std::vector<
  typename decltype(map)::value_type
> vec;
vec.reserve(map.size());
ranges::move(map, std::back_inserter(vec));</pre></td><td><pre>auto vec = get_widgets_map()
          | ranges::to<vector>();</pre></td></tr>
</table>

Design Notes

`from_range` is declared as an instance of a tag_type `from_range_t` in `std`.

`ranges::to`

- `ranges::to` is declared in `<ranges>`
- Can deduce the container value type (using ctad).

- Uses the most efficient construction method (copy constructor, tagged range constructors, iterators constructors, `std::copy`)
- Supports recursion.

Containers range constructors and methods

For any constructor or methods taking a pair of `InputIterator`s in containers (with the exception of `regex` and `filesystem::path`), a similar method is added taking a range instead. All added constructors are tagged with `from_range_t`. Methods that may be ambiguous are suffixed with `_range`.

The container's value type must be explicitly constructible from the reference type of the `input_range` Range.

The following methods and constructors are added to all sequence containers (`vector, deque, list, forward_list`):

- `Container(from_range_t, Range, const Allocator& = {});`
- `iterator insert_range(const_iterator position, Range&&);`
- `void assign(Range&&);`

`basic_string` gains:

- `basic_string(from_range_t, Range, const Allocator& = {});`
- `iterator insert_range(const_iterator position, Range&&);`
- `basic_string& assign(Range&&);`
- `basic_string& replace(const_iterator, const_iterator, Range&&);`
- `basic_string& append(Range&&);`

[*Note:* in `basic_string`, the added `append`, `replace`, `assign` methods do not participate in overload resolution if the range is convertible to string_view. — *end note*]

The following methods and constructors are added to associative containers:

- `Container(from_range_t, Range, const Compare& = {}, const Allocator& = {});`
- `void insert_range(Range&&);`

The following methods and constructors are added to unordered containers:

- `Container(from_range_t, Range, size_t n = /**/, const hasher& = {}, const key_equal& ={}, const Allocator& = {});`
- `Container(from_range_t, Range, size_t n, const Allocator&);`
- `Container(from_range_t, Range, size_t n, const hasher&, const Allocator&);`
- `void insert_range(Range&&);.`

`priority_queue` gains:

- `priority_queue(from_range_t, Range, const Compare& = {});`
- `priority_queue(from_range_t, Range, const Compare, const Alloc&);`
- `priority_queue(from_range_t, Range, const Alloc&);`

`stack` and `queue` gain:

- `Container(from_range_t, Range);`
- `Container(from_range_t, Range, const Alloc&);`

For every constructor, a deduction guide is added.

Considerations

Why do we need this?

Containers do not have containers constructors, so

```
vector v = views::iota(0, 10);
```

is currently not valid syntax.

They do have a constructors taking a pair of iterators. So, it would theorically be possible to write:

```
auto view = views::iota(0, 10);
vector<int> v(ranges::begin(view), ranges::end(view));
```

Which is more cumbersome. But that isn't enough! Containers expect the same types for both iterators - they do not support sentinels. A solution would be to write:

```
auto view = views::iota(0, 10) | views::commom;
vector<int> v(ranges::begin(view), ranges::end(view));
```

And that still does not always work. 'Cpp17Iterators' requiered by containers can have slightly different semantics. Namely, 'input_iterator' may not be copyable. So the following does not work:

```
std::generator<std::pair<int, string>> f() {
   co_yield {0, "Hello"};
   co_yield {1, "World"};
}
auto view = f(); // attempts to use views::common here would be ill-formed
map<int, std::string> v(ranges::begin(view), ranges::end(view));
```

Instead, one has to insert each element manually.

This is sufficiently complex and error-prone approach that multiple blog posts and stackoverflow questions address it:

- How to make a container from a C++20 range
- Will we be able to construct containers with views in C++20?
- Range concept and containers constructors
- Initializing std::vector with ranges library

Why do we need a tag / different method names ?

Ambiguities can arise in 2 cases. Using different methods resolves these ambiguities.

CTAD

Consider the following code:

```
std::list<int> l;
std::vector v{l};
```

Should `v` be `std::vector<int>` or `std::vector<std::list<int>>` ? It is currently equivalent to the latter! Adding a tag solves this issue(although one needs to remember using a tag!).

Ambiguous conversions

They other issues is with `vector<any>`.

```
std::list<any> l;
std::vector<any> v;
v.insert(l);
```

Does that insert a range of any or a single any? Using a different name resolves this ambiguity.

`assign` always takes a range or count + value, so it does not suffer this ambiguity.

Do we need both approaches?

Both approaches are complementary. `ranges::to` works with non-standard containers and supports constructing containers of containers. Tagged constructors offer an opportunity for containers to provide a more efficient implementation. For example, many containers do not have a `reserve` method. `ranges::to` uses the tagged constructor when available. As such, tagged constructors offer a customization mechanism to opt-in into ranges::to and could be extended to things that are not containers.

`ranges::to` does not replace the proposed `insert_range` and `assign` methods.

While we recommend pursuing both approaches, it is essential to make sure `ranges::to` is part of C++23 as it is a critical missing piece of ranges. Approaches can easily be split off if necessary.

Do we need iterator/sentinel pair constructors as well?

The new standard library algorithms in the ‘std::ranges‘ namespace all provide multiple overloads taking both a range and an iterator/sentinel pair. This raises the question: should our new constructors provide a ”range” overload (only), an iterator/sentinel pair overload (only), or both?

The second case (an iterator/sentinel pair only) is easily dismissed. There exist ranges which model `sized_range`, but whose iterators/sentinels do not model `sized_sentinel_for` - the canonical example being `std::list`. With such types, a constructor call like

```
std::list<int> list = get_list();
std::vector<int> vec(std::from_range, list); // copy into vector
```

would know the size of the list and would therefore be able to allocate the required vector size upfront. On the other hand, a hypothetical call to

```
std::list<int> list = get_list();
std::vector<int> vec(std::from_range, list.begin(), list.end());
```

cannot know the size of the input range upfront, and so would need two passes over the data.

We are thus left to decide whether to provide an iterator/sentinel constructor in addition to a ”range” constructor. In the authors’ opinion, this would be redundant. In cases where we do have separate iterator and sentinel objects that we wish to pass to a container constructor, we can do so using ‘subrange‘, as in

```
std::vector vec(std::from_range, subrange(my_iter, my_sentinel));
```

Furthermore, it is intended that users should be able to opt-in to `ranges::to` support for their own container classes by providing constructors which take `from_range_t` as their first argument. Adding iterator/sentinel constructors in addition to ”range” constructors means more work for users to adhere to the protocol, and no doubt risks confusion about whether one, either, or both are required.

Implementation Experience

Implementations of ‘ranges::to‘ are available in [**?**], [**?**] and on Github [**?**]. The tagged ranges constructors, insert methods and other range-taking container members fuctions have **not** been implemented.

Related Paper and future work

Future work is needed to allow constructing `std::array` from *tiny-ranges*.

Acknowledgements

We would like to thank the people who gave feedback on this paper, notably Christopher Di Bella, Arthur O'Dwyer, Barry Revzin and Tristan Brindle.

Proposed wording

Wording is relative to [**?**].

� Containers [containers]

� Container requirements [container.requirements]

� General container requirements [container.requirements.general]

- `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)
  ```

- `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 can choose a different definition. — *end note*]

  The following exposition-only concept is used in the definition of containers

  ```
  template<class T, class R>
  concept compatible_range = // exposition only
        ranges::input_range<R> &&
        constructible_from<T, ranges::range_reference_t<R>> &&
        !convertible_to<ranges::range_reference_t<R>, T>;
  ```

� Sequence containers [sequence.reqmts]

In Tables [tab:container.seq.req] and [tab:container.seq.opt],

- `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 that meet the *Cpp17InputIterator* requirements and refer to elements implicitly convertible to `value_type`,
- `[i, j)` denotes a valid range,
- `range` denotes a value of type `R` such that
  - `R` models `ranges::input_range`,
  - `is_constructible<T, ranges::range_reference_t<R>>` is `true`
- `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&&`.

The complexities of the expressions are sequence dependent.

Table 1: Sequence container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition
`X(n, t)` `X u(n, t);`		*Expects:* `T` is *Cpp17CopyInsertable* into `X`. *Ensures:* `distance(begin(), end()) == n` *Effects:* Constructs a sequence container with `n` copies of `t`

Table 1: Sequence container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition
`X(i, j)` `X u(i, j);`		*Expects:* `T` is *Cpp17EmplaceConstructible* into `X` from `*i`. For `vector`, if the iterator does not meet the *Cpp17ForwardIterator* requirements, `T` is also *Cpp17MoveInsertable* into `X`. *Ensures:* `distance(begin(), end()) == distance(i, j)` *Effects:* Constructs a sequence container equal to the range `[i, j)`. Each iterator in the range `[i, j)` is dereferenced exactly once.
`X(from_range, range)`		*Expects:* `T` is *Cpp17EmplaceConstructible* into `X` from `range_reference_t<R>`. *Effects:* Constructs a sequence container equal to the range `range`. Each iterator in the range `range` is dereferenced exactly once. *Ensures:* `distance(begin(), end()) == ranges::distance(range)`
`X(il)`		Equivalent to `X(il.begin(), il.end())`
`a = il`	`X&`	*Expects:* `T` 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. *Returns:* `*this`.
`a.emplace(p, args)`	`iterator`	*Expects:* `T` is *Cpp17EmplaceConstructible* into `X` from `args`. For `vector` and `deque`, `T` is also *Cpp17MoveInsertable* into `X` and *Cpp17MoveAssignable*. *Effects:* Inserts an object of type `T` constructed with `std::forward<Args>(args)...` before `p`.
`a.insert(p,t)`	`iterator`	*Expects:* `T` is *Cpp17CopyInsertable* into `X`. For `vector` and `deque`, `T` is also *Cpp17CopyAssignable*. *Effects:* Inserts a copy of `t` before `p`.

Table 1: Sequence container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition
`a.insert(p,rv)`	`iterator`	*Expects:* `T` is *Cpp17MoveInsertable* into `X`. For `vector` and `deque`, `T` is also *Cpp17MoveAssignable*. *Effects:* Inserts a copy of `rv` before `p`.
`a.insert(p,n,t)`	`iterator`	*Expects:* `T` is *Cpp17CopyInsertable* into `X` and *Cpp17CopyAssignable*. *Effects:* Inserts `n` copies of `t` before `p`.
`a.insert(p,i,j)`	`iterator`	*Expects:* `T` is *Cpp17EmplaceConstructible* into `X` from `*i`. For `vector` and `deque`, `T` is also *Cpp17MoveInsertable* into `X`, *Cpp17MoveConstructible*, *Cpp17MoveAssignable*, and swappable. Neither `i` nor `j` are iterators into `a`. *Effects:* Inserts copies of elements in `[i, j)` before `p`. Each iterator in the range `[i, j)` shall be dereferenced exactly once.
`a.insert_range, p, range)`	`iterator`	*Expects:* For `vector` and `deque`, `range_reference_t<R>` is *Cpp17MoveInsertable* into `X`, *Cpp17MoveConstructible*, *Cpp17MoveAssignable*, and swappable. `range` and `*this` do not overlap. *Effects:* Inserts copies of elements in `range` before `p`. Each iterator in the range `range` shall be dereferenced exactly once.
`X(il)`		Equivalent to `X(il.begin(), il.end())`
`a.insert(p, il)`	`iterator`	`a.insert(p, il.begin(), il.end())`.
`a.erase(q)`	`iterator`	*Expects:* For `vector` and `deque`, `T` is *Cpp17MoveAssignable*. *Effects:* Erases the element pointed to by `q`.
`a.erase(q1,q2)`	`iterator`	*Expects:* For `vector` and `deque`, `T` is *Cpp17MoveAssignable*. *Effects:* Erases the elements in the range `[q1, q2)`.

Table 1: Sequence container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition
`a.clear()`	`void`	*Effects:* 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()` is `true`. *Complexity:* Linear.
`a.assign(i,j)`	`void`	*Expects:*`T` is *Cpp17EmplaceConstructible* into `X` from `*i` and assignable from `*i`. For `vector`, if the iterator does not meet the forward iterator requirements, `T` is also *Cpp17MoveInsertable* into `X`. Neither `i` nor `j` are iterators into `a`. *Effects:* 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. Each iterator in the range `[i, j)` shall be dereferenced exactly once.
`a.assign(p, range)`	`void`	*Expects:*`range` and `*this` do not overlap. *Effects:* Replaces elements in `a` with a copy of each element in `range`. Invalidates all references, pointers and iterators referring to the elements of `a`. For `vector` and `deque`, also invalidates the past-the-end iterator. Each iterator in the range `range` shall be dereferenced exactly once.
`a.assign(il)`	`void`	`a.assign(il.begin(), il.end())`.
`a.assign(n,t)`	`void`	*Expects:*`T` is *Cpp17CopyInsertable* into `X` and *Cpp17CopyAssignable*. `t` is not a reference into `a`. *Effects:* 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.

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

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

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`.

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`.

The iterator returned from `a.insert_range(p, range)` points to the copy of the first element inserted into `a`, or `p` if `ranges::empty(range)`.

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.

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

� Associative containers [associative.reqmts]

� General [associative.reqmts.general]

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

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.

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.

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.

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>`.

`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*]

The associative containers meet all the requirements of Allocator-aware containers, except that for `map` and `multimap`, the requirements placed on `value_type` in 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*]

In ,

- `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` (),
- `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` meet the *Cpp17InputIterator* requirements and refer to elements implicitly convertible to `value_type`,
- `[i, j)` denotes a valid range,
- `range` denotes a value of type `R` such that
  - `R` models `ranges::input_range`.
  - `is_constructible<typename X::value_type, ranges::range_reference_t<R>>` is `true`.
- `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 2: Associative container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
X(i,j,c) X u(i,j,c);		*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);		*Expects:* key_compare 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(from_range, range, c)		*Effects:* Constructs an empty container and insert each element from range into it. Uses C as the comparison object.	$N \log N$ in general, where N has the value ranges::distance(range); linear if range is sorted with value_comp()
X(from_range, range)		*Expects:* key_compare meets the *Cpp17DefaultConstructible* requirements. *Effects:* Constructs an empty container and insert each element from range into it. Uses Compare() as the comparison object.	same as above

Table 2: Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`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&`	*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()`
`a.insert(i, j)`	`void`	*Expects:*`value_type` is *Cpp17EmplaceConstructible* into `X` from `*i`. Neither `i` nor `j` are iterators into `a`. *Effects:* 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)`

Table 2: Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.insert_range(range)`	`void`	*Expects:* `value_type` is *Cpp17EmplaceConstructible* into `X` from `range_reference_t<R>`. Neither `range` and `a` do not overlap. *Effects:* Inserts each element from `range` 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 `ranges::distance(range)`

The `insert`, `insert_range` 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.

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.

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
```

For associative containers with unique keys the stronger condition holds:

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

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, through either 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.

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.

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

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
- It has a `Compare` template parameter and a type that qualifies as an allocator is deduced for that parameter.

� General [unord.req.general]

// ...

In ,

- `X` denotes an unordered associative container class,
- `a` denotes a value of type `X`,
- `a2` denotes a value of a type with nodes compatible with type `X` (),
- `b` denotes a possibly const value of type `X`,
- `a_uniq` denotes a value of type `X` when `X` supports unique keys,
- `a_eq` denotes a value of type `X` when `X` supports equivalent keys,
- `a_tran` denotes a possibly const value of type `X` when the *qualified-id* s `X::key_equal::is_-transparent` and `X::hasher::is_transparent` are both valid and denote types,
- `i` and `j` denote input iterators that refer to `value_type`,
- `[i, j)` denotes a valid range,
- `range` denotes a value of type `R` such that
  - `R` models `ranges::input_range`.
  - `is_constructible<value_type, ranges::range_reference_t<R>>` is `true`.
- `p` and `q2` denote valid constant iterators to `a`,
- `q` and `q1` denote valid dereferenceable constant iterators to `a`,
- `r` denotes a valid dereferenceable iterator to `a`,
- `[q1, q2)` denotes a valid range in `a`,
- `il` denotes a value of type `initializer_list<value_type>`,

- `t` denotes a value of type `X::value_type`,
- `k` denotes a value of type `key_type`,
- `hf` denotes a possibly const value of type `hasher`,
- `eq` denotes a possibly const value of type `key_equal`,
- `ke` is a value such that
  - `eq(r1, ke) == eq(ke, r1)`
  - `hf(r1) == hf(ke)` if `eq(r1, ke)` is `true`, and
  - `(eq(r1, ke) && eq(r1, r2)) == eq(r2, ke)`

  where `r1` and `r2` are keys of elements in `a_tran`,
- `n` denotes a value of type `size_type`,
- `z` denotes a value of type `float`, and
- `nh` denotes a non-const rvalue of type `X::node_type`.

Table 3: Unordered associative container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X(i, j, n, hf, eq)` `X a(i, j, n, hf, eq);`	`X`	*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 $\mathcal{O}(N^2)$

Table 3: Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X(i, j, n, hf)` `X a(i, j, n, hf);`	`X`	*Expects:*`key_equal` 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`	*Expects:*`hasher` and `key_equal` 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 3: 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`	*Expects:* `hasher` and `key_equal` 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(from_range, range, n, hf, eq)` `X a(from_range, range, n, hf, eq);`	`X`	*Expects:* `value_type` is *Cpp17EmplaceConstructible* into `X` from `range_reference_t<R>`. *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 each element from `range` into it.	Average case $\mathcal{O}(N)$ (N is `ranges::distance(range)`), worst case $\mathcal{O}(N^2)$

Table 3: Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X(from_range, range, n, hf)` `X a(from_range, range, n, hf);`	`X`	*Expects:* `key_equal` meets the *Cpp17DefaultConstructible* requirements. `value_type` is *Cpp17EmplaceConstructible* into `X` from `range_reference_t<R>`. *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 `range` into it.	Average case $\mathscr{O}(N)$ (N is `distance(range)`), worst case $\mathscr{O}(N^2)$
`X(from_range, range, n)` `X a(from_range, range, n);`	`X`	*Expects:* `hasher` and `key_equal` meet the *Cpp17DefaultConstructible* requirements. `value_type` is *Cpp17EmplaceConstructible* into `X` from `range_reference_t<R>`. *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 `range` into it.	Average case $\mathscr{O}(N)$ (N is `ranges::distance(range)`), worst case $\mathscr{O}(N^2)$

Table 3: Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
X(from_range, range) X a(from_range, range);	X	*Expects:* hasher and key_equal meet the *Cpp17DefaultConstructible* requirements. value_type is *Cpp17EmplaceConstructible* into X from range_reference_t<R>. *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 range into it.	Average case $\mathcal{O}(N)$ (N is ranges::distance(range)), worst case $\mathcal{O}(N^2)$
X(il)	X	Same as X(il.begin(), il.end()).	Same as X(il.begin(), il.end()).
a_uniq.insert(t)	pair<iterator, bool>	*Expects:* If t is a non-const rvalue, value_type is *Cpp17MoveInsertable* into X; otherwise, value_type is *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()$)$.

Table 3: Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a_eq.insert(t)`	`iterator`	*Expects:* If `t` is a non-const rvalue, `value_type` is *Cpp17MoveInsertable* into `X`; otherwise, `value_type` is *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`	*Expects:* If `t` is a non-const rvalue, `value_type` is *Cpp17MoveInsertable* into `X`; otherwise, `value_type` is *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`	*Expects:*`value_type` is *Cpp17EmplaceConstructible* into `X` from `*i`. Neither `i` nor `j` are iterators into `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()`$+1))$.
`a.insert_range(range)`	`void`	*Expects:*`value_type` is *Cpp17EmplaceConstructible* into `X` from `range_reference_t<R>`. `range` and `a` are not overlapping. *Effects:* Equivalent to `a.insert(t)` for each element in `range`.	Average case $\mathcal{O}(N)$, where N is `ranges::distance(range)`, worst case $\mathcal{O}(N($`a.size()`$+1))$.

Table 3: 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())`.

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 [1]quality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions. of the partition into equivalent-key groups produced by `Pred`.

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.

The `insert`, `insert_range` 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.

The `insert` ,`insert_range` 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.

[1]E

� Class template deque [deque]

� Overview [deque.overview]

```
namespace std {
    template<class T, class Allocator = allocator<T>>
    class deque {
        public:

        // ??, 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());
        template<compatible_range<T> R>
        deque(from_range_t, R&& range, 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);
        template<compatible_range<T> R>
        void assign(R&& range);
        void assign(size_type n, const T& t);
        void assign(initializer_list<T>);
        allocator_type get_allocator() const noexcept;
        //...
        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);
        template<compatible_range<T> R>
        iterator insert_range(const_iterator position, R&& range);
        iterator insert(const_iterator position, initializer_list<T>);

        //...
    };

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

```
    template<ranges::input_range R, class Allocator = allocator<ranges::range_value_t<R>>>
    deque(R, Allocator = Allocator())
    -> deque<ranges::range_value_t<R>, Allocator>;
}
```

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

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

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

Complexity: Linear in distance(first, last).

```
template<compatible_range<T> R>
deque(from_range_t, R&& range, const Allocator& = Allocator());
```

Effects: Constructs a deque with the elements of the range range, using the specified allocator.

Complexity: Linear in ranges::distance(r).

� 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);
template<compatible_range<T> R>
iterator insert_range(const_iterator position, R&& range);
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);
```

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.

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 at either the beginning or end of a deque always takes constant time and causes a single call to a constructor of T.

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.

� Class template `forward_list` [forwardlist]

� Overview [forwardlist.overview]

```
namespace std {
  template<class T, class Allocator = allocator<T>>
  class forward_list {
    public:

    // ??, 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());
    template<compatible_range<T> R>
    forward_list(from_range_t, R&& range, 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);
    template<compatible_range<T> R>
    void assign(R&& range);
    void assign(size_type n, const T& t);
    void assign(initializer_list<T>);
    allocator_type get_allocator() const noexcept;

    //...

    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);
        template<compatible_range<T> R>
        iterator insert_after_from_range(const_iterator position, R&& range);
        iterator insert_after(const_iterator position, initializer_list<T> il);

    };

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

    template<ranges::input_range R, class Allocator = allocator<ranges::range_value_t<R>>>
    forward_list(R, Allocator = Allocator())
    -> forward_list<ranges::range_value_t<R>, Allocator>;
}
```

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

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

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

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

```
template<compatible_range<T> R>}
forward_list(from_range_t, R&& range, const Allocator& = Allocator());
```

Effects: Constructs a `forward_list` object with the elements of the range `range`.

Complexity: Linear in `ranges::distance(r)`.

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

Expects: `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`. Neither `first` nor `last` are iterators in `*this`.

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

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

```
template<compatible_range<T> R>}
iterator insert_after_from_range(const_iterator position, R&& range);}
```

Expects: `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`. `ranges::begin(first)` is not an iterator in `*this`.

Effects: Inserts copies of elements in the range `[r, a)` fter `position`.

Returns: An iterator pointing to the last inserted element or `position` if `ranges::emty(r)` is `true`.

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

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

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

� Class template `list` [list]

� Overview [list.overview]

```
namespace std {
    template<class T, class Allocator = allocator<T>>
    class list {
        public:

        // ??, 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());
        template<compatible_range<T> R>
        list(from_range_t, R&& range, 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);
        template<compatible_range<T> R>
        void assign(R&& range);
        void assign(size_type n, const T& t);
        void assign(initializer_list<T>);
        allocator_type get_allocator() const noexcept;

        //...

        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);
        template<compatible_range<T> R>
        iterator insert_range(const_iterator position, R&& range);
        iterator insert(const_iterator position, initializer_list<T> il);
```

```
    //...
  };

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

  template<ranges::input_range R, class Allocator = allocator<ranges::range_value_t<R>>>
  list(R, Allocator = Allocator())
  -> list<ranges::range_value_t<R>, Allocator>;
}
```

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

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

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

```
template<compatible_range<T> R>}
list(from_range_t, R&& range, const Allocator& = Allocator());
```

Effects: Constructs a `list` object with the elements of the range `range`.

Complexity: Linear in `ranges::distance(r)`.

� 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<compatible_range<T> R>
  iterator insert_range(const_iterator position, R&& range);

  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);
```

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.

Remarks: Does not affect the validity of iterators and references. If an exception is thrown there are no effects.

� Class template vector [vector]

� Overview [vector.overview]

```
namespace std {
    template<class T, class Allocator = allocator<T>>
    class vector {
        public:
        // ??, construct/copy/destroy
        constexpr vector() noexcept(noexcept(Allocator())) : vector(Allocator()) { }
        constexpr explicit vector(const Allocator&) noexcept;
        constexpr explicit vector(size_type n, const Allocator& = Allocator());
        constexpr vector(size_type n, const T& value, const Allocator& = Allocator());
        template<class InputIterator>
        constexpr vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
        template<compatible_range<T> R>
        constexpr vector(from_range_t, R&& range, const Allocator& = Allocator());
        constexpr vector(const vector& x);
        constexpr vector(vector&&) noexcept;
        constexpr vector(const vector&, const Allocator&);
        constexpr vector(vector&&, const Allocator&);
        constexpr vector(initializer_list<T>, const Allocator& = Allocator());
        constexpr ~vector();
        constexpr vector& operator=(const vector& x);
        constexpr vector& operator=(vector&& x)
        noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
        allocator_traits<Allocator>::is_always_equal::value);
        constexpr vector& operator=(initializer_list<T>);
        template<class InputIterator>
        constexpr void assign(InputIterator first, InputIterator last);
        template<compatible_range<T> R>
        constexpr void assign(R&& range);
        constexpr void assign(size_type n, const T& u);
        constexpr void assign(initializer_list<T>);
        constexpr allocator_type get_allocator() const noexcept;

        //...

        template<class... Args> constexpr iterator emplace(const_iterator position, Args&&... args);
        constexpr iterator insert(const_iterator position, const T& x);
        constexpr iterator insert(const_iterator position, T&& x);
        constexpr iterator insert(const_iterator position, size_type n, const T& x);
        template<class InputIterator>
        constexpr iterator insert(const_iterator position, InputIterator first, InputIterator last);
        template<compatible_range<T> R>
        constexpr iterator insert_range(const_iterator position, R&& range);
        constexpr iterator insert(const_iterator position, initializer_list<T> il);
        constexpr iterator erase(const_iterator position);
```

```
        constexpr iterator erase(const_iterator first, const_iterator last);
        constexpr void     swap(vector&)
        noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
        allocator_traits<Allocator>::is_always_equal::value);
        constexpr void     clear() noexcept;
    };

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

    template<ranges::input_range R, class Allocator = allocator<ranges::range_value_t<R>>>
    vector(R, Allocator = Allocator())
    -> vector<ranges::range_value_t<R>, Allocator>;
}
```

� Constructors [vector.cons]

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

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

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.

```
template<compatible_range<T> R>}
vector(from_range_t, R&& range, const Allocator& = Allocator());
```

Effects: Constructs a `vector` object with the elements of the range `range`, using the specified allocator.

Complexity: Makes only N calls to the constructor of `T` (where N is `ranges::distance(range)` and no reallocations if `range` models `ranges::forward_range` or `ranges::sized_range`. Otherwise, it makes order `N` calls to the constructor of `T` and order $\log N$ reallocations.

� Modifiers [vector.modifiers]

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

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

Complexity: If reallocation happens, linear in the number of elements of the resulting vector; otherwise, linear in the number of elements inserted plus the distance to the end of the vector.

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, as well as the past-the-end iterator. If no reallocation happens, then references, pointers, and iterators before the insertion point remain valid but those at or after the insertion point, including the past-the-end iterator, are invalidated. 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.

� Class `vector<bool>` [vector.bool]

```
namespace std {
    template<class Allocator>
    class vector<bool, Allocator> {
        public:
        //...
        // construct/copy/destroy
        constexpr vector() : vector(Allocator()) { }
        constexpr explicit vector(const Allocator&);
        constexpr explicit vector(size_type n, const Allocator& = Allocator());
        constexpr vector(size_type n, const bool& value, const Allocator& = Allocator());
        template<class InputIterator>
        constexpr vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
        template<compatible_range<bool> R>
        constexpr vector(from_range_t, R&& range, const Allocator& = Allocator());
        constexpr vector(const vector& x);
        constexpr vector(vector&& x);
        constexpr vector(const vector&, const Allocator&);
        constexpr vector(vector&&, const Allocator&);
        constexpr vector(initializer_list<bool>, const Allocator& = Allocator());
        constexpr ~vector();
        constexpr vector& operator=(const vector& x);
        constexpr vector& operator=(vector&& x);
        constexpr vector& operator=(initializer_list<bool>);
        template<class InputIterator>
        constexpr void assign(InputIterator first, InputIterator last);
        template<compatible_range<bool> R>
        constexpr void assign(R&& range);
```

```
    constexpr void assign(size_type n, const bool& t);
    constexpr void assign(initializer_list<bool>);
    constexpr allocator_type get_allocator() const noexcept;
    //...

    // modifiers
    template<class... Args> constexpr reference emplace_back(Args&&... args);
    constexpr void push_back(const bool& x);
    constexpr void pop_back();
    template<class... Args> constexpr iterator emplace(const_iterator position, Args&&... args);
    constexpr iterator insert(const_iterator position, const bool& x);
    constexpr iterator insert(const_iterator position, size_type n, const bool& x);
    template<class InputIterator>
    constexpr iterator insert(const_iterator position, InputIterator first, InputIterator last);
    template<compatible_range<bool> R>
    constexpr iterator insert_range(const_iterator position, R&& range);
    constexpr iterator insert(const_iterator position, initializer_list<bool> il);
  };
}
```

Associative containers [associative]

Class template `map` [map]

Overview [map.overview]

```
namespace std {
template<class Key, class T, class Compare = less<Key>,
class Allocator = allocator<pair<const Key, T>>>
class map {
  public:

  // ??, 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());

  template<compatible_range<value_type> R>
  map(from_range_t, R&& range, 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) { }

    template<compatible_range<value_type> R>
    map(from_range_t, R&& range, const Allocator& a))
    : map(from_range, std::forward<R>(range), Compare(), a){}

    map(initializer_list<value_type> il, const Allocator& a)
    : map(il, Compare(), a) { }

    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);
    template<compatible_range<value_type> R>
    void insert_range(R&& range);

    void insert(initializer_list<value_type>);

   //...
};

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<ranges::input_range R,
    class Compare = less<iter-key-type<ranges::iterator_t<R>>>,
    class Allocator = allocator<iter-to-alloc-type<ranges::iterator_t<R>>>
map(from_range_t, R, Compare = Compare(), Allocator = Allocator())
    -> map<iter-key-type<ranges::iterator_t<R>>,
           iter-mapped-type<ranges::iterator_t<R>>,
           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<ranges::input_range R, class Allocator>
```

```
map(from_range_t, R, Allocator)
-> map<iter-key-type<ranges::iterator_t<R>>,
    iter-mapped-type<ranges::iterator_t<R>>,
    less<iter-key-type<ranges::iterator_t<R>>>, Allocator>;
```

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

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

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

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

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

```
template<compatible_range<value_type> R>}
map(from_range_t, R&& range, const Compare& comp = Compare(), const Allocator& = Allocator());
```

Effects: Constructs an empty `map` using the specified comparison object and allocator, and inserts elements from the range `range`.

Complexity: Linear in N if `range` is already sorted using `comp` and otherwise $N \log N$, where N is `ranges::distance(first, last)`.

.

� Class template `multimap` [multimap]

� Overview [multimap.overview]

```
namespace std {
template<class Key, class T, class Compare = less<Key>,
class Allocator = allocator<pair<const Key, T>>>
class multimap {

    // ??, 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& = Al
    template<compatible_range<value_type> R>
    multimap(from_range_t, R&& range, 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) { }

    template<compatible_range<value_type> R>
    multimap(from_range_t, R&& range, const Allocator& a))
    : multimap(from_range, std::forward<R>(range), Compare(), a){}

    multimap(initializer_list<value_type> il, const Allocator& a)
    : multimap(il, Compare(), a) { }
    ~multimap();

    // ??, 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);
    template<compatible_range<value_type> R>
    void insert_range(R&& range);
    void insert(initializer_list<value_type>);
};

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<ranges::input_range R,
    class Compare = less<iter-key-type<ranges::iterator_t<R>>>,
    class Allocator = allocator<iter-to-alloc-type<ranges::iterator_t<R>>>
multimap(from_range_t, R, Compare = Compare(), Allocator = Allocator())
-> multimap<iter-key-type<ranges::iterator_t<R>>,
    iter-mapped-type<ranges::iterator_t<R>>,
    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<ranges::input_range R, class Allocator>
multimap(from_range_t, R, Allocator = Allocator())
-> multimap<iter-key-type<ranges::iterator_t<R>>,
    iter-mapped-type<ranges::iterator_t<R>>,
    less<iter-key-type<ranges::iterator_t<R>>>, Allocator>;


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

� Constructors [multimap.cons]

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

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

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

```
template<compatible_range<value_type> R>}
multimap(from_range_t, R&& range, const Compare& comp = Compare(), const Allocator& = Allocator());
```

Effects: Constructs an empty `multimap` using the specified comparison object and allocator, and inserts elements from the range `range`.

Complexity: Linear in N if `range` is already sorted using `comp` and otherwise $N \log N$, where N is `ranges::distance(first, last)`.

.

� Class template `set` [set]

� Overview [set.overview]

```
namespace std {
    template<class Key, class Compare = less<Key>,
    class Allocator = allocator<Key>>
    class set {
        public:
        // ??, 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& = Allo
    template<compatible_range<value_type> R>
    set(from_range_t, R&& range, 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) { }
    template<compatible_range<value_type> R>
    set(from_range_t, R&& range, const Allocator& a))
    : set(from_range, std::forward<R>(range), Compare(), a){}


    set(initializer_list<value_type> il, const Allocator& a)
    : set(il, Compare(), a) { }
    ~set();
    //...
    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);

    template<compatible_range<value_type> R>
    void insert_range(R&& range);

    void insert(initializer_list<value_type>);

};

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<ranges::input_range R, class Compare = less<ranges::range_value_t<R>>,
    class Allocator = allocator<ranges::range_value_t<R>>>
set(from_range_t, R, Compare = Compare(), Allocator = Allocator())
-> set<ranges::range_value_t<R>,  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<ranges::input_range R,  class Allocator>
    set(from_range_t, R, Allocator)
    -> set<ranges::range_value_t<R>,  less<ranges::range_value_t<R>>, Allocator>;


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

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

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

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

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

```
template<compatible_range<value_type> R>}
set(from_range_t, R&& range, const Compare& comp = Compare(), const Allocator& = Allocator());
```

Effects: Constructs an empty `set` using the specified comparison object and allocator, and inserts elements from the range `range`.

Complexity: Linear in N if `range` is already sorted using `comp` and otherwise $N \log N$, where N is `ranges::distance(first, last)`.

.

� Class template `multiset` [multiset]

� Overview [multiset.overview]

```
namespace std {
template<class Key, class Compare = less<Key>,
class Allocator = allocator<Key>>
class multiset {
    public:
    // ??, 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());

    template<compatible_range<value_type> R>
    multiset(from_range_t, R&& range, 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) { }

   template<compatible_range<value_type> R>
   multiset(from_range_t, R&& range, const Allocator& a))
   : multiset(from_range, std::forward<R>(range), Compare(), a){}

    multiset(initializer_list<value_type> il, const Allocator& a)
    : multiset(il, Compare(), a) { }
    ~multiset();

    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);
    template<compatible_range<value_type> R>
    void insert_range(R&& range);

    void insert(initializer_list<value_type>);

};

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<ranges::input_range R, class Compare = less<ranges::range_value_t<R>>,
    class Allocator = allocator<ranges::range_value_t<R>>>
multiset(from_range_t, R, Compare = Compare(), Allocator = Allocator())
-> multiset<ranges::range_value_t<R>,  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<ranges::input_range R,  class Allocator>
multiset(from_range_t, R, Allocator)
-> multiset<ranges::range_value_t<R>,  less<ranges::range_value_t<R>>, Allocator>;


template<class Key, class Allocator>
multiset(initializer_list<Key>, Allocator) -> multiset<Key, less<Key>, Allocator>;
}
```

� Constructors [multiset.cons]

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

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

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

```
template<compatible_range<value_type> R>}
set(from_range_t, R&& range, const Compare& comp = Compare(), const Allocator& = Allocator());
```

Effects: Constructs an empty `multiset` using the specified comparison object and allocator, and inserts elements from the range `range`.

Complexity: Linear in N if `range` is already sorted using `comp` and otherwise $N \log N$, where N is `ranges::distance(first, last)`.

.

� Unordered associative containers [unord]

� Class template `unordered_map` [unord.map]

� Overview [unord.map.overview]

```
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 {
    // ??, 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());

    template<compatible_range<value_type> R>
    unordered_map(from_range_t, R&& range, 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) { }

    template<compatible_range<value_type> R>
    unordered_map(from_range_t, R&& range, size_type n, const allocator_type& a)
    : unordered_map(from_range, forward<R>(range), n, hasher(), key_equal(), a) {}

    template<compatible_range<value_type> R>
    unordered_map(from_range_t, R&& range, size_type n, const hasher& hf, const allocator_type&
a)
    : unordered_map(from_range, forward<R>(range), 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();

    // ??, 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);
    template<compatible_range<value_type> R>
    void insert_range(R&& range);
    void insert(initializer_list<value_type>);

};

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<ranges::input_range R,
    class Hash = hash<iter-key-type<ranges::range_iterator_t<R>>>,
    class Pred = equal_to<iter-key-type<ranges::range_iterator_t<R>>>,
    class Allocator = allocator<iter-to-alloc-type<ranges::range_iterator_t<R>>>>
    >
unordered_map(from_range_t, R, typename see below::size_type = see below,
    Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_map<iter-key-type<ranges::range_iterator_t<R>>, iter-mapped-type<ranges::range_iterator_t<R>>,
    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<ranges::input_range R, class Allocator>
unordered_map(from_range_t, R, typename see below::size_type, Allocator)
-> unordered_map<iter-key-type<ranges::range_iterator_t<R>>,
        iter-mapped-type<ranges::range_iterator_t<R>>,
        hash<iter-key-type<ranges::range_iterator_t<R>>>,
        equal_to<iter-key-type<ranges::range_iterator_t<R>>>, Allocator>;
template<ranges::input_range R, class Allocator>

unordered_map(from_range_t, R, Allocator)
-> unordered_map<iter-key-type<ranges::range_iterator_t<R>>,
    iter-mapped-type<ranges::range_iterator_t<R>>,
    hash<iter-key-type<ranges::range_iterator_t<R>>>,
    equal_to<iter-key-type<ranges::range_iterator_t<R>>>, Allocator>;
template<ranges::input_range R, class Hash, class Allocator>

unordered_map(from_range_t, R, typename see below::size_type, Hash, Allocator)
-> unordered_map<iter-key-type<ranges::range_iterator_t<R>>,
    iter-mapped-type<ranges::range_iterator_t<R>>,
    Hash,
    equal_to<iter-key-type<ranges::range_iterator_t<R>>>, 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>;
}
```

� Constructors [unord.map.cnstr]

```
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());
template<compatible_range<value_type> R>
unordered_map(from_range_t, R&& range, 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());
```

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 `range` from the second form, or from the range `[il.begin(), il.end())` for the ~~second~~ third form. `max_load_factor()` returns `1.0`.

Complexity: Average case linear, worst case quadratic.

� Class template `unordered_multimap` [unord.multimap]

� Overview [unord.multimap.overview]

```
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:
    // ??, 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());
```

```
template<compatible_range<value_type> R>
unordered_multimap(from_range_t, R&& range, 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) { }

template<compatible_range<value_type> R>
unordered_multimap(from_range_t, R&& range, size_type n, const allocator_type& a)
: unordered_multimap(from_range, forward<R>(range), n, hasher(), key_equal(), a) {}

template<compatible_range<value_type> R>
unordered_multimap(from_range_t, R&& range, size_type n, const hasher& hf, const allocator_type&
a)
: unordered_multimap(from_range, forward<R>(range), 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) { }


// ??, 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);
    template<compatible_range<value_type> R>
    void insert_range(R&& range);
    void insert(initializer_list<value_type>);

};

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<ranges::input_range R,
    class Hash = hash<iter-key-type<ranges::range_iterator_t<R>>>,
    class Pred = equal_to<iter-key-type<ranges::range_iterator_t<R>>>,
    class Allocator = allocator<iter-to-alloc-type<ranges::range_iterator_t<R>>>>
    >
unordered_multimap(from_range_t, R, typename see below::size_type = see below,
Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_multimap<iter-key-type<ranges::range_iterator_t<R>>,
    iter-mapped-type<ranges::range_iterator_t<R>>,
    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<ranges::input_range R, class Allocator>
unordered_multimap(from_range_t, R, typename see below::size_type, Allocator)
-> unordered_multimap<iter-key-type<ranges::range_iterator_t<R>>,
    iter-mapped-type<ranges::range_iterator_t<R>>,
    hash<iter-key-type<ranges::range_iterator_t<R>>>,
    equal_to<iter-key-type<ranges::range_iterator_t<R>>>, Allocator>;
    template<ranges::input_range R, class Allocator>

template<ranges::input_range R, class Allocator>
unordered_multimap(from_range_t, R, Allocator)
-> unordered_multimap<iter-key-type<ranges::range_iterator_t<R>>,
    iter-mapped-type<ranges::range_iterator_t<R>>,
    hash<iter-key-type<ranges::range_iterator_t<R>>>,
    equal_to<iter-key-type<ranges::range_iterator_t<R>>>, Allocator>;
    template<ranges::input_range R, class Hash, class Allocator>

template<ranges::input_range R, class Hash, class Allocator>
unordered_multimap(from_range_t, R, typename see below::size_type, Hash, Allocator)
-> unordered_multimap<iter-key-type<ranges::range_iterator_t<R>>,
    iter-mapped-type<ranges::range_iterator_t<R>>,
    Hash,
    equal_to<iter-key-type<ranges::range_iterator_t<R>>>, 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>;
}
```

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.

� Constructors [unord.multimap.cnstr]

```
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());
template<compatible_range<value_type> R>
unordered_multimap(from_range_t, R&& range, 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());
```

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 `range` from the second form, or from the range `[il.begin(), il.end())` for the ~~second~~ third form. `max_load_factor()` returns `1.0`.

Complexity: Average case linear, worst case quadratic.

� Class template `unordered_set` [unord.set]

� Overview [unord.set.overview]

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

    // ??, 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());

    template<compatible_range<T> R>
    unordered_set(from_range_t, R&& range, 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) { }

    template<compatible_range<T> R>
    unordered_set(from_range_t, R&& range, size_type n, const allocator_type& a)
    : unordered_set(from_range, forward<R>(range), n, hasher(), key_equal(), a) {}

    template<compatible_range<T> R>
    unordered_set(from_range_t, R&& range, size_type n, const hasher& hf, const allocator_type&
a)
    : unordered_set(from_range, forward<R>(range), 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) { }

    // 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);
    template<compatible_range<value_type> R>
    void insert_range(R&& range);
    void insert(initializer_list<value_type>);

};

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<ranges::input_range R,
    class Hash = hash<ranges::range_value_t<R>>,
    class Pred = equal_to<ranges::range_value_t<R>>,
    class Allocator = allocator<ranges::range_value_t<R>>>
unordered_set(from_range_t, R, typename see below::size_type = see below,
    Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_set<ranges::range_value_t<R>>, 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<ranges::input_range R, class Allocator>
unordered_set(from_range_t, R, typename see below::size_type, Allocator)
-> unordered_set<ranges::range_value_t<R>,
    hash<ranges::range_value_t<R>>,
    equal_to<ranges::range_value_t<R>>, Allocator>;
    template<ranges::input_range R, class Allocator>

template<ranges::input_range R, class Allocator>
unordered_set(from_range_t, R, Allocator)
-> unordered_set<ranges::range_value_t<R>,
    hash<ranges::range_value_t<R>>,
    equal_to<ranges::range_value_t<R>>, Allocator>;
    template<ranges::input_range R, class Hash, class Allocator>

template<ranges::input_range R, class Hash, class Allocator>
```

```
unordered_set(from_range_t, R, typename see below::size_type, Hash, Allocator)
-> unordered_set<ranges::range_value_t<R>,
    Hash,
    equal_to<ranges::range_value_t<R>>, 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>;
}
```

```
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());
template<compatible_range<T> R>
unordered_multiset(from_range_t, R&& range, 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());
```

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 `range` from the second form, or from the range `[il.begin(), il.end())` for the ~~second~~ third form. `max_load_factor()` returns `1.0`.

Complexity: Average case linear, worst case quadratic.

� Class template `unordered_multiset` [unord.multiset]

� Overview [unord.multiset.overview]

```
namespace std {
template<class Key,
class Hash = hash<Key>,
class Pred = equal_to<Key>,
class Allocator = allocator<Key>>
class unordered_multiset {
    public:
    // types
```

```
// ??, 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());

template<compatible_range<T> R>
unordered_multiset(from_range_t, R&& range, 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) { }

template<compatible_range<T> R>
unordered_multiset(from_range_t, R&& range, size_type n, const allocator_type& a)
: unordered_multiset(from_range, forward<R>(range), n, hasher(), key_equal(), a) {}

template<compatible_range<T> R>
unordered_multiset(from_range_t, R&& range, size_type n, const hasher& hf, const allocator_type&
a)
: unordered_multiset(from_range, forward<R>(range), 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();

    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);
    template<compatible_range<value_type> R>
    void insert_range(R&& range);
    void insert(initializer_list<value_type>);

};

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<ranges::input_range R,
    class Hash = hash<ranges::range_value_t<R>>,
    class Pred = equal_to<ranges::range_value_t<R>>,
    class Allocator = allocator<ranges::range_value_t<R>>>
unordered_multiset(from_range_t, R, typename see below::size_type = see below,
    Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_multiset<ranges::range_value_t<R>>, 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<ranges::input_range R, class Allocator>
unordered_multiset(from_range_t, R, typename see below::size_type, Allocator)
-> unordered_multiset<ranges::range_value_t<R>,
    hash<ranges::range_value_t<R>>,
    equal_to<ranges::range_value_t<R>>, Allocator>;
    template<ranges::input_range R, class Allocator>

template<ranges::input_range R, class Allocator>
unordered_multiset(from_range_t, R, Allocator)
-> unordered_multiset<ranges::range_value_t<R>,
    hash<ranges::range_value_t<R>>,
    equal_to<ranges::range_value_t<R>>, Allocator>;
    template<ranges::input_range R, class Hash, class Allocator>

template<ranges::input_range R, class Hash, class Allocator>
unordered_multiset(from_range_t, R, typename see below::size_type, Hash, Allocator)
-> unordered_multiset<ranges::range_value_t<R>,
    Hash,
    equal_to<ranges::range_value_t<R>>, 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>;
}
```

� Constructors [unord.multiset.cnstr]

```
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());
template<compatible_range<T> R>
unordered_multiset(from_range_t, R&& range, 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());
```

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 `range` from the second form or from the range `[il.begin(), il.end())` for the ~~second~~ third form. `max_load_factor()` returns `1.0`.

Complexity: Average case linear, worst case quadratic.

� Container adaptors [container.adaptors]

The wording for the [container.adaptors] section assumes P1425 has been accepted

� Definition [queue.defn]

```
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 InputIterator>
        queue(InputIterator first, InputIterator last);
        template<compatible_range<T> R>
        queue(from_range_t, R&& range);

        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&);

        template<class InputIterator, class Alloc>
        queue(InputIterator first, InputIterator last, const Alloc&);
        template<compatible_range<T> R, class Alloc>
        queue(from_range_t, R&& range, const Alloc&);

        //...
    };
```

```
    template<class Container>
    queue(Container) -> queue<typename Container::value_type, Container>;

    template<class InputIterator>
    queue(InputIterator, InputIterator) -> queue<iter-value-type<InputIterator>>;


    template<ranges::input_range R>
    queue(from_range_t, R)
    -> queue<ranges::range_value_t<R>>;

    template<class Container, class Allocator>
    queue(Container, Allocator) -> queue<typename Container::value_type, Container>;

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

    template<ranges::input_range R, class Allocator = allocator<ranges::range_value_t<R>>>
    queue(from_range_t, R, Allocator)
    -> queue<ranges::range_value_t<R>, Allocator>;

    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 { };
}
```

� Constructors [queue.cons]

```
explicit queue(const Container& cont);
```

Effects: Initializes `c` with `cont`.

```
explicit queue(Container&& cont);
```

Effects: Initializes `c` with `std::move(cont)`.

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

Effects: Initializes `c` with `first` as the first argument and `last` as the second argument.

```
template<compatible_range<T> R>
queue(from_range_t, R&& range);
```

Effects: Initializes `c` with `ranges::to<Container>(std::forward<R>(range))`.

� Constructors with allocators [queue.cons.alloc]

[...]

```
template<class Alloc> queue(const queue& q, const Alloc& a);
```

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);
```

Effects: Initializes `c` with `std::move(q.c)` as the first argument and `a` as the second argument.

```
template<class InputIterator, class Alloc>
queue(InputIterator first, InputIterator last, const Alloc & alloc);
```

Effects: Initializes `c` with `first` as the first argument, `last` as the second argument and `alloc` as the third argument.

```
template<compatible_range<T> R, class Alloc>
queue(from_range_t, R&& range, const Alloc& a);
```

Effects: Initializes `c` with `ranges::to<Container>(std::forward<R>(range), a)`.

� Class template `priority_queue` [priority.queue]

� Overview [priqueue.overview]

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` and `deque` 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<compatible_range<T> R>
    priority_queue(from_range_t, R&& range, const Compare& x = Compare());


    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&);

    template<compatible_range<T> R, class Alloc>
    priority_queue(from_range_t, R&& range, const Compare&, const Alloc&)
    template<compatible_range<T> R, class Alloc>
    priority_queue(from_range_t, R&& range, 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<ranges::input_range R,class Compare = less<ranges::range_value_t<R>>
priority_queue(from_range_t, R, Compare = Compare())
```

```
-> priority_queue<ranges::range_value_t<R>, vector<ranges::range_value_t<R>>, 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>;

template<ranges::input_range R,
 class Compare = less<ranges::range_value_t<R>,
 class Allocator = allocator<ranges::range_value_t<R>>
priority_queue(from_range_t, R, Compare, Allocator)
 -> priority_queue<ranges::range_value_t<R>,
 vector<ranges::range_value_t<R>, Allocator>, Compare>;

template<ranges::input_range R, class Allocator = allocator<ranges::range_value_t<R>>
priority_queue(from_range_t, R, Allocator)
-> priority_queue<ranges::range_value_t<R>,
 vector<ranges::range_value_t<R>, Allocator>, less<ranges::range_value_t<R>>;

// no equality is provided

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

� Constructors [priqueue.cons]

```
priority_queue(const Compare& x, const Container& y);
priority_queue(const Compare& x, Container&& y);
```

Expects: `x` defines a strict weak ordering.

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());
```

Expects: `x` defines a strict weak ordering.

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)`.

```
template<compatible_range<T> R>
priority_queue(from_range_t, R&& range, const Compare& x = Compare());
```

Expects: `x` defines a strict weak ordering.

Effects: Initializes `comp` with `x` and `c` with `ranges::to<Container>(std::forward<R>(range))` and finally calls `make_heap(c.begin(), c.end(), comp)`.

� Constructors with allocators [priqueue.cons.alloc]

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);
```

Effects: Initializes `c` with `a` and value-initializes `comp`.

```
template<class Alloc> priority_queue(const Compare& compare, const Alloc& a);
```

Effects: Initializes `c` with `a` and initializes `comp` with `compare`.

```
template<class Alloc>
priority_queue(const Compare& compare, const Container& cont, const Alloc& a);
```

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);
```

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);
```

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);
```

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)`.

```
template<compatible_range<T> R, class Alloc>
priority_queue(from_range_t, R&& range, const Compare& compare, const Alloc& a);
```

Effects: initializes `comp` with `compare` and `c` with `ranges::to<Container>(std::forward<R>(range, a))` ; calls `make_heap(c.begin(), c.end(), comp)`.

```
template<compatible_range<T> R, class Alloc>
priority_queue(from_range_t, R&& range, const Alloc& a);
```

Effects: Initializes `c` with `ranges::to<Container>(std::forward<R>(range, a));` calls `make_heap(c.begin(), c.end(), comp)`.

� Class template stack [stack]

� 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 InputIterator>
        stack(InputIterator first, InputIterator last);
        template<compatible_range<T> R>
        stack(from_range_t, R&& range);

        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&);


        template<class InputIterator, class Alloc>
        stack(InputIterator first, InputIterator last, const Alloc&);
        stack<compatible_range<T> R, class Alloc>
        stack(from_range_t, R&& range, const Alloc&);

        //...
    };

    template<class Container>
    stack(Container) -> stack<typename Container::value_type, Container>;
```

```
    template<class InputIterator>
    stack(InputIterator, InputIterator)
    -> stack<iter-value-type<InputIterator>>;

    template<ranges::input_range R>
    stack(from_range_t, R)
    -> stack<ranges::range_value_t<R>>;

    template<class Container, class Allocator>
    stack(Container, Allocator) -> stack<typename Container::value_type, Container>;

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

    template<ranges::input_range R, class Allocator = allocator<ranges::range_value_t<R>>>
    stack(from_range_t, R, Allocator)
    -> stack<ranges::range_value_t<R>, Allocator>;


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

� Constructors [stack.cons]

```
explicit stack(const Container& cont);
```

Effects: Initializes `c` with `cont`.

```
explicit stack(Container&& cont);
```

Effects: Initializes `c` with `std::move(cont)`.

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

Effects: Initializes `c` with `first` as the first argument and `last` as the second argument.

```
template<compatible_range<T> R>
stack(from_range_t, R&& range);
```

Effects: Initializes `c` with `ranges::to<Container>(std::forward<R>(range))`.

� Constructors with allocators [stack.cons.alloc]

[...]

```
template<class Alloc> stack(const stack& s, const Alloc& a);
```

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);
```

Effects: Initializes `c` with `std::move(s.c)` as the first argument and `a` as the second argument.

```
template<class InputIterator, class Alloc>
stack(InputIterator first, InputIterator last, const Alloc& alloc);
```

Effects: Initializes `c` with `first` as the first argument, `last` as the second argument and `alloc` as the third argument.

```
template<compatible_range<T> R, class Alloc>
stack(from_range_t, R&& range, const Alloc& a);
```

Effects: Initializes `c` with `ranges::to<Container>(std::forward<R>(range), a)`.

� Class template `basic_string` [basic.string]

```
namespace std {
    template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
    class basic_string {
        public:

        // ??, construct/copy/destroy
        constexpr basic_string() noexcept(noexcept(Allocator())) : basic_string(Allocator()) { }
        constexpr explicit basic_string(const Allocator& a) noexcept;
        constexpr basic_string(const basic_string& str);
        constexpr basic_string(basic_string&& str) noexcept;
        constexpr basic_string(const basic_string& str, size_type pos,
        const Allocator& a = Allocator());
        constexpr basic_string(const basic_string& str, size_type pos, size_type n,
        const Allocator& a = Allocator());
        template<class T>
        constexpr basic_string(const T& t, size_type pos, size_type n,
        const Allocator& a = Allocator());
        template<class T>
        constexpr explicit basic_string(const T& t, const Allocator& a = Allocator());
        constexpr basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
        constexpr basic_string(const charT* s, const Allocator& a = Allocator());
        constexpr basic_string(size_type n, charT c, const Allocator& a = Allocator());
        template<class InputIterator>
        constexpr basic_string(InputIterator begin, InputIterator end, const Allocator& a = Allocator());

        template<ranges::input_range R>
        requires constructible_from<charT, ranges::range_reference_t<R>
        constexpr basic_string(from_range_t, R&& range, const Allocator& a = Allocator());

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

```
constexpr ~basic_string();

// ??, modifiers
constexpr basic_string& operator+=(const basic_string& str);
template<class T>
constexpr basic_string& operator+=(const T& t);
constexpr basic_string& operator+=(const charT* s);
constexpr basic_string& operator+=(charT c);
constexpr basic_string& operator+=(initializer_list<charT>);
constexpr basic_string& append(const basic_string& str);
constexpr basic_string& append(const basic_string& str, size_type pos, size_type n = npos);
template<class T>
constexpr basic_string& append(const T& t);
template<class T>
constexpr basic_string& append(const T& t, size_type pos, size_type n = npos);
constexpr basic_string& append(const charT* s, size_type n);
constexpr basic_string& append(const charT* s);
constexpr basic_string& append(size_type n, charT c);
template<class InputIterator>
constexpr basic_string& append(InputIterator first, InputIterator last);

template<ranges::input_range R>
requires constructible_from<charT, ranges::range_reference_t<R>>
constexpr basic_string& append(R&& range);

constexpr basic_string& append(initializer_list<charT>);

constexpr void push_back(charT c);

constexpr basic_string& assign(const basic_string& str);
constexpr basic_string& assign(basic_string&& str)
noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
allocator_traits<Allocator>::is_always_equal::value);
constexpr basic_string& assign(const basic_string& str, size_type pos, size_type n = npos);
template<class T>
constexpr basic_string& assign(const T& t);
template<class T>
constexpr basic_string& assign(const T& t, size_type pos, size_type n = npos);
constexpr basic_string& assign(const charT* s, size_type n);
constexpr basic_string& assign(const charT* s);
constexpr basic_string& assign(size_type n, charT c);
template<class InputIterator>
constexpr basic_string& assign(InputIterator first, InputIterator last);

template<ranges::input_range R>
requires constructible_from<charT, ranges::range_reference_t<R>>
constexpr basic_string& assign(R&& range);

constexpr basic_string& assign(initializer_list<charT>);

constexpr basic_string& insert(size_type pos, const basic_string& str);
```

```
constexpr basic_string& insert(size_type pos1, const basic_string& str,
size_type pos2, size_type n = npos);
template<class T>
constexpr basic_string& insert(size_type pos, const T& t);
template<class T>
constexpr basic_string& insert(size_type pos1, const T& t,
size_type pos2, size_type n = npos);
constexpr basic_string& insert(size_type pos, const charT* s, size_type n);
constexpr basic_string& insert(size_type pos, const charT* s);
constexpr basic_string& insert(size_type pos, size_type n, charT c);
constexpr iterator insert(const_iterator p, charT c);
constexpr iterator insert(const_iterator p, size_type n, charT c);
template<class InputIterator>
constexpr iterator insert(const_iterator p, InputIterator first, InputIterator last);

template<ranges::input_range R>
requires constructible_from<charT, ranges::range_reference_t<R>
constexpr iterator insert_range(const_iterator p, R&& range);

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

//...

constexpr basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
constexpr basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
size_type pos2, size_type n2 = npos);
template<class T>
constexpr basic_string& replace(size_type pos1, size_type n1, const T& t);
template<class T>
constexpr basic_string& replace(size_type pos1, size_type n1, const T& t,
size_type pos2, size_type n2 = npos);
constexpr basic_string& replace(size_type pos, size_type n1, const charT* s, size_type n2);
constexpr basic_string& replace(size_type pos, size_type n1, const charT* s);
constexpr basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);
constexpr basic_string& replace(const_iterator i1, const_iterator i2,
const basic_string& str);
template<class T>
constexpr basic_string& replace(const_iterator i1, const_iterator i2, const T& t);
constexpr basic_string& replace(const_iterator i1, const_iterator i2, const charT* s,
size_type n);
constexpr basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);
constexpr basic_string& replace(const_iterator i1, const_iterator i2, size_type n, charT c);
template<class InputIterator>
constexpr basic_string& replace(const_iterator i1, const_iterator i2,
    InputIterator j1, InputIterator j2);

template<ranges::input_range R>
requires constructible_from<charT, ranges::range_reference_t<R>
constexpr basic_string& replace_with_range(const_iterator i1, const_iterator i2, R&& range);

constexpr basic_string& replace(const_iterator, const_iterator, initializer_list<charT>);
```

```
    //....
};

template<class InputIterator,
class Allocator = allocator<typename iterator_traits<InputIterator>::value_type>>
basic_string(InputIterator, InputIterator, Allocator = Allocator())
-> basic_string<typename iterator_traits<InputIterator>::value_type,
char_traits<typename iterator_traits<InputIterator>::value_type>,
Allocator>;


template<ranges::input_range R, class Allocator = allocator<ranges::range_value_t<R>>>
basic_string(R, Allocator = Allocator())
-> basic_string<ranges::range_value_t<R>, char_traits<ranges::range_value_t<R>>, Allocator>;

template<class charT,
class traits,
class Allocator = allocator<charT>>
explicit basic_string(basic_string_view<charT, traits>, const Allocator& = Allocator())
-> basic_string<charT, traits, Allocator>;

template<class charT,
class traits,
class Allocator = allocator<charT>>
basic_string(basic_string_view<charT, traits>,
typename see below::size_type, typename see below::size_type,
const Allocator& = Allocator())
-> basic_string<charT, traits, Allocator>;
}
```

� Constructors and assignment operators [string.cons]

```
template<class InputIterator>
constexpr basic_string(InputIterator begin, InputIterator end, const Allocator& a = Allocator());
```

Constraints: `InputIterator` is a type that qualifies as an input iterator.

Effects: Constructs a string from the values in the range `[begin, end)`, as indicated in .

```
template<ranges::input_range R>
requires constructible_from<charT, ranges::range_reference_t<R>
basic_string(from_range_t, R&& range, const Allocator& = Allocator());
```

Effects: Constructs a string from the values in the range `range` as indicated in [container.seq.req].

� Modifiers [string.modifiers]

� basic_string::operator+= [string.op.append]

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

Constraints: `InputIterator` is a type that qualifies as an input iterator.

Effects: Equivalent to: `return append(basic_string(first, last, get_allocator()));`

```
template<ranges::input_range R>
requires constructible_from<charT, ranges::range_reference_t<R>>
constexpr basic_string& append(R&& range);
```

Constraints:

- `is_convertible_v<R, const CharT*>` is `false`, and
- `is_convertible_v<R, basic_string_view<CharT, Traits>>` is `false`.

Effects: Equivalent to: `return append(basic_string(from_range, forward<R>(range), get_allocator()));`

� `basic_string::assign` [string.assign]

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

Constraints: `InputIterator` is a type that qualifies as an input iterator.

Effects: Equivalent to: `return assign(basic_string(first, last, get_allocator()));`

```
template<ranges::input_range R>
requires constructible_from<charT, ranges::range_reference_t<R>>
constexpr basic_string& assign(R&& range);
```

Constraints:

- `is_convertible_v<R, const CharT*>` is `false`, and
- `is_convertible_v<R, basic_string_view<CharT, Traits>>` is `false`.

Effects: Equivalent to: `return assign(basic_string(from_range, forward<R>(range), get_allocator()));`

� `basic_string::insert` [string.insert]

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

Constraints: `InputIterator` is a type that qualifies as an input iterator.

Expects: `p` is a valid iterator on `*this`.

Effects: Equivalent to `insert(p - begin(), basic_string(first, last, get_allocator()))`.

Returns: An iterator which refers to the first inserted character, or `p` if `first == last`.

```
template<ranges::input_range R>
requires constructible_from<charT, ranges::range_reference_t<R>
constexpr iterator insert_range(const_iterator p, R&& range);
```

Expects: `p` is a valid iterator on `*this`.

Effects: Equivalent to `insert(p - begin(), basic_string(from_range, forward<R>(range), get_allocator()))`.

Returns: An iterator which refers to the first inserted character, or `p` if `ranges::empty(range)`.

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

Effects: Equivalent to: `return insert(p, il.begin(), il.end());`

� `basic_string::replace` [string.replace]

```
template<class InputIterator>
constexpr basic_string& replace(const_iterator i1, const_iterator i2,
InputIterator j1, InputIterator j2);
```

Constraints: `InputIterator` is a type that qualifies as an input iterator.

Effects: Equivalent to: `return replace(i1, i2, basic_string(j1, j2, get_allocator()));`

```
template<ranges::input_range R>
requires constructible_from<charT, ranges::range_reference_t<R>
constexpr basic_string& replace(const_iterator i1, const_iterator i2, R&& range);
```

Constraints:

- `is_convertible_v<R, const CharT*>` is `false`, and
- `is_convertible_v<R, basic_string_view<CharT, Traits>>` is `false`.

Effects: Equivalent to: `return replace(i1, i2, basic_string(from_range, forward<R>(range), get_allocator()));`

```
constexpr basic_string& replace(const_iterator i1, const_iterator i2, initializer_list<charT> il);
```

Effects: Equivalent to: `return replace(i1, i2, il.begin(), il.size());`

� Header `<ranges>` synopsis [ranges.syn]

```
#include <compare>              // see ??
#include <initializer_list>     // see ??
#include <iterator>             // see ??

namespace std::ranges {
    template<class R>
    using keys_view = elements_view<views::all_t<R>, 0>;
    template<class R>
```

```
    using values_view = elements_view<views::all_t<R>, 1>;

    namespace views {
        template<size_t N>
        inline constexpr unspecified elements = unspecified ;
        inline constexpr auto keys = elements<0>;
        inline constexpr auto values = elements<1>;
    }

    template <class C, input_range R, class... Args>
    requires (!view<C>)
    constexpr C to(R&& r, Args&&... args);

    template <template <class...> class C, input_range R, class... Args>
    constexpr auto to(R && r, Args&&... args) -> see below;
}
namespace std {
    namespace views = ranges::views;

    template<class I, class S, ranges::subrange_kind K>
    struct tuple_size<ranges::subrange<I, S, K>>
    : integral_constant<size_t, 2> {};
    template<class I, class S, ranges::subrange_kind K>
    struct tuple_element<0, ranges::subrange<I, S, K>> {
        using type = I;
    };
    template<class I, class S, ranges::subrange_kind K>
    struct tuple_element<1, ranges::subrange<I, S, K>> {
        using type = S;
    };
    template<class I, class S, ranges::subrange_kind K>
    struct tuple_element<0, const ranges::subrange<I, S, K>> {
        using type = I;
    };
    template<class I, class S, ranges::subrange_kind K>
    struct tuple_element<1, const ranges::subrange<I, S, K>> {
        using type = S;
    };

    struct from_range_t;
    inline constexpr from_range_t from_range;
}
```

In [range.utility] Insert after section [range.dangling]

� Range conversions [range.utility.conversions]

� Range argument tag [range.utility.conversions.tag]

```
namespace std {
    struct from_range_t { explicit from_range_t() = default; };
```

```
    inline constexpr from_range_t from_range{};
}
```

The from_range_t struct is an empty class type used as a unique type to disambiguate constructor and function overloading. Specifically, several types, notably containers have constructors with from_range_t as the first argument>

The range conversions functions efficiently construct an instance of type from a range. ranges::to<C>(r, args) returns an instance c of C constructed by the first valid method among the following:

- Construct c from r
- Construct c from the pair of iterators ranges::begin(r), ranges::end(r)
- Construct c, then insert each element of r at the end of c.
- If C is a range whose value type is itself a range (and is not a view), and r's value type is also a range, the application of to<range_value_t<C>> for each element of r is inserted at the end of c.

ranges::to [range.utility.conversions.adaptor]

When the instance c of C in constructed, the parameter pack args is forwarded as the trailing parameters of the selected constructor of C. This allows passing an allocator to the selected constructor.

```
template <class Container, class R>
concept reservable-container =  // exposition only
    sized_range<R> &&
    requires(Container& c, R&& r) {
    { c.capacity(); } -> same_as<range_size_t<C>>;
    { c.reserve(c.capacity()); };
};


template <class C, input_range R, class... Args>
requires (!view<C>)
constexpr C to(R&& r, Args&&... args);
```

Returns an instance of C constructed from the elements of r in the following manner:

- If constructible_from<C, R, Args...> is true, equivalent to C(std::forward<R>(r), std::forward<Args>(args)...).
- Otherwise, if constructible_from<C, from_range_t, R, Args...> is true, equivalent to C(from_range, std::forward<R>(r), std::forward<Args>(args)...).
- Otherwise, if
  - constructible_from<C, Args...> is true,
  - indirectly_copyable<iterator_t<R>, iterator_t<C>> is true, and
  - inserter(c, ranges::end(c)) is a valid expression.

equivalent to:

```
C c(std::forward<Args...>(args)...);
if constexpr (reservable-container<C, R>) {
    c.reserve(ranges::size(r));
}
ranges::copy(r, inserter(c, ranges::end(c)));
```

- Otherwise, if:
  - `input_range<range_value_t<C>>` is `true`,
  - `input_range<range_value_t<R>>` is `true`,
  - `view<range_value_t<C>>` is `false`,
  - `indirectly_copyable<`
    `iterator_t<range_reference_t<R>>,`
    `iterator_t<range_value_t<C>>`
    `>` is `true`, and
  - `inserter(c, ranges::end(c))` is a valid expression.

  equivalent to:

```
C c(std::forward<Args...>(args)...);
if constexpr (reservable-container<C, R>) {
    c.reserve(ranges::size(r));
}
auto v = r | views::transform ([](auto&& elem) {
    return to<range_value_t<C>>(elem);
});
ranges::copy(v, inserter(c, ranges::end(c)));
```

- Otherwise `ranges::to<C>(r, args)` is ill-formed.

```
template <template <class...> class C, input_range R, class... Args>
constexpr auto to(R&& r, Args&&... args) -> DEDUCE_TYPE(R);
```

Let `ITER` be a type meeting the requirements of `Cpp17InputIterator` such that

- *`ITER`*`::iterator_category` is `input_iterator_tag`,
- *`ITER`*`::value_type` is `range_value_t<R>`,
- *`ITER`*`::difference_type` is `range_difference_t<R>`,
- *`ITER`*`::pointer` is `add_pointer_t<range_reference_t<R>>`, and
- *`ITER`*`::reference` is `range_reference_t<R>`.

Let *DEDUCE_TYPE(D)* be defined as follows:

- `decltype(C(declval<D>(), declval<Args>()...))` if that is a valid expression,

- Otherwise, `decltype(C(from_range, declval<D>(), declval<Args>()...))` if that is a valid expression,
- Otherwise, `decltype(C(declval<ITER>(), declval<ITER>(), declval<Args>()...))` if that is a valid expression
- Otherwise, `DEDUCE_TYPE(D)` is ill-formed.

Mandates:

- `DEDUCE_TYPE(R)` denotes a type.

Effects:

- Expression-equivalent to `ranges::to<DEDUCE_TYPE(R)>(r, args...)`.

� `ranges::to` adaptors [range.utility.conversions.adapters]

In addition to the functions described above, `ranges::to` also defines a closure object that accepts a `viewable_range` argument and returns an instance of a type `T` such that the expressions `r | ranges::to<T>(args...)` and `ranges::to<T>(r, args...)` have equivalent semantics. [*Note:* T denotes either a class or a class template — *end note*] .

The bitwise OR operator is overloaded for the purpose of chaining `ranges::to` to the end of an adaptor chain pipeline.

[*Example:*

```
list<int> ints{0,1,2,3,4,5};

auto v1 = ints | ranges::to<vector>();
auto v2 = ints | ranges::to<vector<int>>();
auto v3 = ranges::to<vector>(ints);
auto v4 = ranges::to<vector<int>>(ints);

assert(v1 == v2 && v2 == v3 && v3 == v4);
```

— *end example*]

Feature test macros

Bump the value of `__cpp_lib_ranges` to the date of adoption.

Add a new macro in `<version>` `__cpp_lib_containers_ranges` set to the date of adoption.

The macro `__cpp_lib_containers_ranges` is also present in <vector>, <list>, <forward_list> <map>, <set> <unordered_map>, <unordered_set>, <deque>, <queue>, <priority_queue>, <stack>, and <string>

Implementation Experience

Implementations of ‘ranges::to‘ are available in [**?**], [**?**] and on Github [**?**]. The tagged ranges constructors, insert methods and other range-taking container members fuctions have **not** been implemented.

Related Paper and future work

Future work is needed to allow constructing `std::array` from *tiny-ranges*.

Acknowledgements

We would like to thank the people who gave feedback on this paper, notably Christopher Di Bella, Arthur O’Dwyer, Barry Revzin and Tristan Brindle.

References

[cmcstl2] https://github.com/CaseyCarter/cmcstl2/blob/a7a714a9159b08adeb00a193e77b782846b3b20e/include/stl2/detail/to.hpp

[RangeV3] Eric Niebler https://github.com/ericniebler/range-v3/blob/v1.0-beta/include/range/v3/to_container.hpp

[rangesnext] Corentin Jabot https://github.com/cor3ntin/rangesnext/blob/master/include/cor3ntin/rangesnext/to.hpp

[CTAD Ranges] Eric Niebler https://github.com/ericniebler/range-v3/blob/d284e9c84ff69bb416d9d94d029729dfb38c3364/include/range/v3/range/conversion.hpp#L140-L152

[P1391] Corentin Jabot *Range constructor for std::string_view*
https://wg21.link/P1391

[N4885] Thomas Köppe *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4885