ISO/IEC JTC1 SC22 WG21 P2416R0
Author: Jens Maurer
Target audience: LWG
2021-07-15

P2416R0: Presentation of requirements in the standard library

Introduction

This paper suggests a change in presentation of the requirements tables in the standard library.

The existing tables are awkward and frequently do not use established best practice for presenting requirements.

The following pages present the container and regular expression requirements in a new format for comment. No semantic changes are intended.

Acknowledgements

Thanks to the project (co)editors for review and assistance.

22 Containers library [containers]

22.1 General [containers.general]

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

² The following subclauses describe container requirements, and components for sequence containers and associative containers, as summarized in Table 72.

Table 72: Containers library summary   [tab:containers.summary]

	Subclause	Header
22.2	Requirements	
22.3	Sequence containers	`<array>`, `<deque>`, `<forward_list>`, `<list>`, `<vector>`
22.4	Associative containers	`<map>`, `<set>`
22.5	Unordered associative containers	`<unordered_map>`, `<unordered_set>`
22.6	Container adaptors	`<queue>`, `<stack>`
22.7	Views	`<span>`

22.2 Requirements [container.requirements]

22.2.1 Preamble [container.requirements.pre]

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

² All of the complexity requirements in this Clause are stated solely in terms of the number of operations on the contained objects.

[*Example 1*: The copy constructor of type `vector<vector<int>>` has linear complexity, even though the complexity of copying each contained `vector<int>` is itself linear. — *end example*]

³ For the components affected by subclause 22.2 that declare an `allocator_type`, objects stored in these components shall be constructed using the function `allocator_traits<allocator_type>::rebind_traits<U>::construct` and destroyed using the function `allocator_traits<allocator_type>::rebind_traits<U>::destroy` (20.10.8.3), where U is either `allocator_type::value_type` or an internal type used by the container. These functions are called only for the container's element type, not for internal types used by the container.

[*Note 1*: This means, for example, that a node-based container would need to construct nodes containing aligned buffers and call `construct` to place the element into the buffer. — *end note*]

22.2.2 General containers [container.gen.reqmts]

22.2.2.1 General [container.requirements.general]

¹ In subclause 22.2.2,

(1.1) — `X` denotes a container class containing objects of type `T`,

(1.2) — `a` and `b` denote values of type `X`,

(1.3) — `i` and `j` denote values of type (possibly const) `X::iterator`,

(1.4) — `u` denotes an identifier,

(1.5) — `r` denotes a non-const value of type `X`, and

(1.6) — `rv` denotes a non-const rvalue of type `X`.

22.2.2.2 Containers [container.reqmts]

¹ A type `X` meets the *container* requirements if the following types and expressions are well-formed and have the specified semantics.

(1.1) — The name `X::value_type` denotes the type `T`. `T` is *Cpp17Erasable* from `X` (see 22.2.2.1, below).

(1.2) — The name `X::reference` denotes the type `T&`.

(1.3) — The name `X::const_reference` denotes the type `const T&`.

(1.4) — The name `X::iterator` denotes a type that meets the forward iterator requirements (23.3.5.5) with value type `T`. The type `X::iterator` is convertible to `X::const_iterator`.

(1.5) — The name `X::const_iterator` denotes a type that meets the requirements of a constant iterator and those of a forward iterator with value type `T`.

(1.6) — The name `X::difference_type` denotes a signed integer type, identical to the difference type of `X::iterator` and `X::const_iterator`.

(1.7) — The name `X::size_type` denotes an unsigned integer type that can represent any non-negative value of `difference_type`.

`X u;` and `X u = X();` are statements, each with the following properties:

2   *Postconditions*: `u.empty()`

3   *Complexity*: Constant.

`X u(a);` and `X u = a;` are statements, each with the following properties:

4   *Preconditions*: `T` is *Cpp17CopyInsertable* into `X` (see below).

5   *Postconditions*: `u == a`

6   *Complexity*: Linear.

`X u(rv);` and `X u = rv;` are statements, each with the following properties:

7   *Postconditions*: `u` is equal to the value that `rv` had before this construction.

8   *Complexity*: Linear for `array` and constant for all other standard containers.

`a = rv`   is an expression with the following properties:

9   *Result*: An lvalue of type `X`.

10   *Effects*: All existing elements of `a` are either move assigned to or destroyed.

11   *Postconditions*: If `a` and `rv` do not refer to the same object, `a` is equal to the value that `rv` had before this assignment.

12   *Complexity*: Linear.

`a.~X()`   is an expression with the following properties:

13   *Result*: `void`

14   *Effects*: Destroys every element of `a`; any memory obtained is deallocated.

15   *Complexity*: Linear.

`a.begin()`   is an expression with the following properties:

16   *Result*: A prvalue of type `iterator`; `const_iterator` for constant `a`.

17   *Value*: An iterator referring to the first element in the container.

18   *Complexity*: Constant.

`a.end()`   is an expression with the following properties:

19   *Result*: A prvalue of type `iterator`; `const_iterator` for constant `a`.

20   *Value*: An iterator which is the past-the-end value for the container.

21   *Complexity*: Constant.

`a.cbegin()`   is an expression with the following properties:

22   *Result*: A prvalue of type `const_iterator`.

23   *Value*: `const_cast<X const&>(a).begin()`

24   *Complexity*: Constant.

`a.cend()` is an expression with the following properties:

25   *Result*: A prvalue of type `const_iterator`.

26   *Value*: `const_cast<X const&>(a).end()`

27   *Complexity*: Constant.

`i <=> j` is an expression with the following properties:

28   *Result*: A prvalue of type `strong_ordering`.

29   *Constraints*: `X::iterator` meets the random access iterator requirements.

30   *Complexity*: Constant.

`a == b` is an expression with the following properties:

31   *Preconditions*: `T` meets the *Cpp17EqualityComparable* requirements

32   *Result*: Convertible to `bool`.

33   *Value*: `equal(a.begin(), a.end(), b.begin(), b.end())`
     [*Note 1*: The algorithm `equal` is defined in 25.6.11. — *end note*]

34   *Complexity*: Constant if `a.size() != b.size()`, linear otherwise.

35   *Remarks*: `==` is an equivalence relation.

`a != b` is an expression with the following properties:

36   *Effects*: Equivalent to `!(a == b)`.

`a.swap(b)` is an expression with the following properties:

37   *Result*: `void`

38   *Effects*: Exchanges the contents of `a` and `b`.

39   *Complexity*: Linear for `array` and constant for all other standard containers.

`swap(a, b)` is an expression with the following properties:

40   *Effects*: Equivalent to `a.swap(b)`.

`r = a` is an expression with the following properties:

41   *Result*: An lvalue of type `X`.

42   *Postconditions*: `r == a`.

43   *Complexity*: Linear.

`a.size()` is an expression with the following properties:

44   *Result*: A prvalue of type `size_type`.

45   *Value*: `distance(a.begin(), a.end())`, i.e. the number of elements in the container.

46   *Complexity*: Constant.

47   *Remarks*: The number of elements is defined by the rules of constructors, inserts, and erases.

`a.max_size()` is an expression with the following properties:

48   *Result*: A prvalue of type `size_type`.

49   *Returns*: `distance(begin(), end())` for the largest possible container.
     Complexity: Constant.

`a.empty()`

50   *Result*: Convertible to `bool`.

51   *Value*: `a.begin() == a.end()`

52   *Complexity*: Constant.

53   *Remarks*: If the container is empty, then `a.empty()` is true.

54 In the expressions

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

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

55 Unless otherwise specified, all containers defined in this Clause obtain memory using an allocator (see 16.4.4.6).

[*Note 2*: In particular, containers and iterators do not store references to allocated elements other than through the allocator's pointer type, i.e., as objects of type P or `pointer_traits<P>::template rebind<`*unspecified*`>`, where P is `allocator_traits<allocator_type>::pointer`. — *end note*]

Copy constructors for these container types obtain an allocator by calling `allocator_traits<allocator_-type>::select_on_container_copy_construction` on the allocator belonging to the container being copied. Move constructors obtain an allocator by move construction from the allocator belonging to the container being moved. Such move construction of the allocator shall not exit via an exception. All other constructors for these container types take a `const allocator_type&` argument.

[*Note 3*: If an invocation of a constructor uses the default value of an optional allocator argument, then the allocator type must support value-initialization. — *end note*]

A copy of this allocator is used for any memory allocation and element construction performed, by these constructors and by all member functions, during the lifetime of each container object or until the allocator is replaced. The allocator may be replaced only via assignment or `swap()`. Allocator replacement is performed by copy assignment, move assignment, or swapping of the allocator only if

(55.1) — `allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value`,

(55.2) — `allocator_traits<allocator_type>::propagate_on_container_move_assignment::value`, or

(55.3) — `allocator_traits<allocator_type>::propagate_on_container_swap::value`

is `true` within the implementation of the corresponding container operation. In all container types defined in this Clause, the member `get_allocator()` returns a copy of the allocator used to construct the container or, if that allocator has been replaced, a copy of the most recent replacement.

56 The expression `a.swap(b)`, for containers `a` and `b` of a standard container type other than `array`, shall exchange the values of `a` and `b` without invoking any move, copy, or swap operations on the individual container elements. Lvalues of any `Compare`, `Pred`, or `Hash` types belonging to `a` and `b` shall be swappable and shall be exchanged by calling `swap` as described in 16.4.4.3. If `allocator_traits<allocator_type>::propagate_-on_container_swap::value` is `true`, then lvalues of type `allocator_type` shall be swappable and the allocators of `a` and `b` shall also be exchanged by calling `swap` as described in 16.4.4.3. Otherwise, the allocators shall not be swapped, and the behavior is undefined unless `a.get_allocator() == b.get_-allocator()`. Every iterator referring to an element in one container before the swap shall refer to the same element in the other container after the swap. It is unspecified whether an iterator with value `a.end()` before the swap will have value `b.end()` after the swap.

22.2.2.3 Reversible container requirements [container.rev.reqmts]

1 A type X meets the *reversible container* requirements if X meets the container requirements, the iterator type of X belongs to the bidirectional or random access iterator categories (23.3), and the following types and expressions are well-formed and have the specified semantics.

(1.1) — The name `X::reverse_iterator` denotes the type `reverse_iterator<X::iterator>`, an iterator type whose value type is T.

(1.2) — The name `X::const_reverse_iterator` denotes the type `reverse_iterator<X::const_iterator>`, a constant iterator type whose value type is T.

`a.rbegin()`   is an expression with the following properties:

2   *Result*: `reverse_iterator`; `const_reverse_iterator` for constant `a`

3   *Value*: `reverse_iterator(end())`

4   *Complexity*: Constant.

`a.rend()`   is an expression with the following properties:

5   *Result*: `reverse_iterator`; `const_reverse_iterator` for constant `a`

6   *Value*: `reverse_iterator(begin())`

7   *Complexity*: Constant.

`a.crbegin()`   is an expression with the following properties:

8   *Result*: `const_reverse_iterator`

9   *Value*: `const_cast<X const&>(a).rbegin()`

10   *Complexity*: Constant.

`a.crend()`   is an expression with the following properties:

11   *Result*: `const_reverse_iterator`

12   *Value*: `const_cast<X const&>(a).rend()`

13   *Complexity*: Constant.

14   Unless otherwise specified (see 22.2.7.2, 22.2.8.2, 22.3.8.4, and 22.3.11.5) all container types defined in this Clause meet the following additional requirements:

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

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

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

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

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

(14.6)   — no `swap()` function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped.

[*Note 1*: The `end()` iterator does not refer to any element, so it can be invalidated. — *end note*]

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

16   A *contiguous container* is a container whose member types `iterator` and `const_iterator` meet the *Cpp17RandomAccessIterator* requirements (23.3.5.7) and model `contiguous_iterator` (23.3.4.14).

22.2.2.4   Optional container requirements   [container.opt.reqmts]

1   The following operations are provided for some types of containers but not others. Those containers for which the listed operations are provided shall implement the semantics as described unless otherwise stated. If the iterators passed to `lexicographical_compare_three_way` meet the constexpr iterator requirements (23.3.1) then the operations described below are implemented by constexpr functions.

`a <=> b`   is an expression with the following properties:

2   *Result*: A prvalue of type *synth-three-way-result*`<X::value_type>`.

3   *Preconditions*: Either `<=>` is defined for values of type (possibly const) `T`, or `<` is defined for values of type (possibly const) `T` and `<` is a total ordering relationship.

4   *Value*: `lexicographical_compare_three_way(a.begin(), a.end(), b.begin(), b.end(),` *synth-three-way*`)`

[*Note 1*: The algorithm `lexicographical_compare_three_way` is defined in Clause 25. — *end note*]

5   *Complexity*: Linear.

22.2.2.5   Allocator-aware containers [container.alloc.reqmts]

¹ All of the containers defined in Clause 22 and in 21.3.3 except `array` meet the additional requirements of an *allocator-aware container*, as described below.

² Given an allocator type `A` and given a container type `X` having a `value_type` identical to `T` and an `allocator_-type` identical to `allocator_traits<A>::rebind_alloc<T>` and given an lvalue `m` of type `A`, a pointer `p` of type `T*`, an expression `v` of type (possibly `const`) `T`, and an rvalue `rv` of type `T`, the following terms are defined. If `X` is not allocator-aware, the terms below are defined as if `A` were `allocator<T>` — no allocator object needs to be created and user specializations of `allocator<T>` are not instantiated:

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

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

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

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

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

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

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

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

   [*Note 1*: `rv` remains a valid object. Its state is unspecified — *end note*]

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

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

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

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

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

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

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

   [*Note 2*: 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*]

³ In this subclause,

(3.1)   — `X` denotes an allocator-aware container class with a `value_type` of `T` using an allocator of type `A`,

(3.2)   — `u` denotes a variable,

(3.3)   — `a` and `b` denote non-const lvalues of type `X`,

(3.4)   — `c` denotes an lvalue of type `const X`,

(3.5)   — `t` denotes an lvalue or a const rvalue of type `X`,

(3.6)   — `rv` denotes a non-const rvalue of type `X`, and

(3.7)   — `m` is a value of type `A`.

A type `X` meets the allocator-aware container requirements if `X` meets the container requirements and the following types and expressions are well-formed and have the specified semantics.

(3.8)   — `X::allocator_type` denotes the type `A`. `allocator_type::value_type` is the same as `X::value_type`.

`c.get_allocator()`   is an expression with the following properties:

⁴   *Result*: `A`

⁵   *Complexity*: Constant.

`X u;` and `X u = X();` are statements, each with the following properties:

⁶   *Preconditions*: `A` meets the *Cpp17DefaultConstructible* requirements.

7   *Postconditions*: `u.empty()` returns `true`, `u.get_allocator() == A()`.

8   *Complexity*: Constant.

`X u(m);`   is a statement with the following properties:

9   *Postconditions*: `u.empty()` returns `true`, `u.get_allocator() == m`.

10 *Complexity*: Constant.

`X u(t, m);`   is a statement with the following properties:

11 *Preconditions*: `T` is *Cpp17CopyInsertable* into `X`.

12 *Postconditions*: `u == t`, `u.get_allocator() == m`

13 *Complexity*: Linear.

`X u(rv);`   is a statement with the following properties:

14 *Postconditions*: `u` has the same elements as `rv` had before this construction; the value of `u.get_-allocator()` is the same as the value of `rv.get_allocator()` before this construction.

15 *Complexity*: Constant.

`X u(rv, m);`   is a statement with the following properties:

16 *Preconditions*: `T` is *Cpp17MoveInsertable* into `X`.

17 *Postconditions*: `u` has the same elements, or copies of the elements, that `rv` had before this construction, `u.get_allocator() == m`.

18 *Complexity*: Constant if `m == rv.get_allocator()`, otherwise linear.

`a = t`   is an expression with the following properties:

19 *Result*: An lvalue of type `X`.

20 *Preconditions*: `T` is *Cpp17CopyInsertable* into `X` and *Cpp17CopyAssignable*.

21 *Postconditions*: `a == t` is `true`.

22 *Complexity*: Linear.

`a = rv`   is an expression with the following properties:

23 *Result*: An lvalue of type `X`.

24 *Preconditions*: If `allocator_traits<allocator_type>::propagate_on_container_move_assign-ment::value` is `false`, `T` is *Cpp17MoveInsertable* into `X` and *Cpp17MoveAssignable*.

25 *Effects*: All existing elements of `a` are either move assigned to or destroyed.

26 *Postconditions*: If `a` and `rv` do not refer to the same object, `a` is equal to the value that `rv` had before this assignment.

27 *Complexity*: Linear.

`a.swap(b)`   is an expression with the following properties:

28 *Result*: `void`

29 *Effects*: Exchanges the contents of `a` and `b`.

30 *Complexity*: Constant.

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

(31.1) — The *qualified-id* `A::value_type` is valid and denotes a type (13.10.3).

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

22.2.3   Container data races   **[container.requirements.dataraces]**

1   For purposes of avoiding data races (16.4.6.10), implementations shall consider the following functions to be `const`: `begin`, `end`, `rbegin`, `rend`, `front`, `back`, `data`, `find`, `lower_bound`, `upper_bound`, `equal_range`, `at` and, except in associative or unordered associative containers, `operator[]`.

2   Notwithstanding 16.4.6.10, implementations are required to avoid data races when the contents of the contained object in different elements in the same container, excepting `vector<bool>`, are modified concurrently.

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

22.2.4   Sequence containers   **[sequence.reqmts]**

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

2   [*Note 1*: The sequence containers offer the programmer different complexity trade-offs. `vector` is appropriate in most circumstances. `array` has a fixed size known during translation. `list` or `forward_list` support frequent insertions and deletions from the middle of the sequence. `deque` supports efficient insertions and deletions taking place at the beginning or at the end of the sequence. When choosing a container, remember `vector` is best; leave a comment to explain if you choose from the rest! — *end note*]

3   In this subclause,

(3.1)   — `X` denotes a sequence container class,

(3.2)   — `a` denotes a value of type `X` containing elements of type `T`,

(3.3)   — `u` denotes the name of a variable being declared,

(3.4)   — `A` denotes `X::allocator_type` if the *qualified-id* `X::allocator_type` is valid and denotes a type (13.10.3) and `allocator<T>` if it doesn't,

(3.5)   — `i` and `j` denote iterators that meet the *Cpp17InputIterator* requirements and refer to elements implicitly convertible to `value_type`,

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

(3.7)   — `il` designates an object of type `initializer_list<value_type>`,

(3.8)   — `n` denotes a value of type `X::size_type`,

(3.9)   — `p` denotes a valid constant iterator to `a`,

(3.10)   — `q` denotes a valid dereferenceable constant iterator to `a`,

(3.11)   — `[q1, q2)` denotes a valid range of constant iterators in `a`,

(3.12)   — `t` denotes an lvalue or a const rvalue of `X::value_type`, and

(3.13)   — `rv` denotes a non-const rvalue of `X::value_type`.

(3.14)   — `Args` denotes a template parameter pack;

(3.15)   — `args` denotes a function parameter pack with the pattern `Args&&`.

4   The complexities of the expressions are sequence dependent.

5   A type `X` meets the *sequence container* requirements if `X` meets the container requirements and the following statements and expressions are well-formed and have the specified semantics.

    `X u(n, t);`   is a statement with the following properties:

6   *Preconditions*: `T` is *Cpp17CopyInsertable* into `X`.

7   *Effects*: Constructs a sequence container with `n` copies of `t`.

8   *Postconditions*: `distance(u.begin(), u.end()) == n` is `true`.

X u(i, j);   is a statement with the following properties:

9   *Preconditions*: T is *Cpp17EmplaceConstructible* into X from *i. For `vector`, if the iterator does not meet the *Cpp17ForwardIterator* requirements (23.3.5.5), T is also *Cpp17MoveInsertable* into X.

10   *Effects*: Constructs a sequence container equal to the range [i, j). Each iterator in the range [i, j) is dereferenced exactly once.

11   *Postconditions*: `distance(u.begin(), u.end()) == distance(i, j)` is `true`.

X(il)   is an expression with the following properties:

12   *Effects*: Equivalent to X(il.begin(), il.end()).

a = il   is an expression with the following properties:

13   *Result*: An lvalue of type X.

14   *Preconditions*: T is *Cpp17CopyInsertable* into X and *Cpp17CopyAssignable*.

15   *Effects*: Assigns the range [il.begin(), il.end()) into a. All existing elements of a are either assigned to or destroyed.

16   *Returns*: *this.

a.emplace(p, args)   is an expression with the following properties:

17   *Result*: A prvalue of type `iterator`.

18   *Preconditions*: T is *Cpp17EmplaceConstructible* into X from args. For `vector` and `deque`, T is also *Cpp17MoveInsertable* into X and *Cpp17MoveAssignable*.

19   *Effects*: Inserts an object of type T constructed with `std::forward<Args>(args)...` before p.

     [*Note 2*: args can directly or indirectly refer to a value in a. — *end note*]

20   *Returns*: An iterator that points to the new element constructed from args into a.

a.insert(p, t)   is an expression with the following properties:

21   *Result*: A prvalue of type `iterator`.

22   *Preconditions*: T is *Cpp17CopyInsertable* into X. For `vector` and `deque`, T is also *Cpp17CopyAssignable*.

23   *Effects*: Inserts a copy of t before p.

24   *Returns*: An iterator that points to the copy of t inserted into a.

a.insert(p, rv)   is an expression with the following properties:

25   *Result*: A prvalue of type `iterator`.

26   *Preconditions*: T is *Cpp17MoveInsertable* into X. For `vector` and `deque`, T is also *Cpp17MoveAssignable*.

27   *Effects*: Inserts a copy of rv before p.

28   *Returns*: An iterator that points to the copy of rv inserted into a.

a.insert(p, n, t)   is an expression with the following properties:

29   *Result*: A prvalue of type `iterator`.

30   *Preconditions*: T is *Cpp17CopyInsertable* into X and *Cpp17CopyAssignable*.

31   *Effects*: Inserts n copies of t before p.

32   *Returns*: An iterator that points to the copy of the first element inserted into a, or p if n == 0.

a.insert(p, i, j)   is an expression with the following properties:

33   *Result*: A prvalue of type `iterator`.

34   *Preconditions*: T is *Cpp17EmplaceConstructible* into X from *i. For `vector` and `deque`, T is also *Cpp17MoveInsertable* into X, *Cpp17MoveConstructible*, *Cpp17MoveAssignable*, and swappable (16.4.4.3). Neither i nor j are iterators into a.

35   *Effects*: Inserts copies of elements in [i, j) before p. Each iterator in the range [i, j) shall be dereferenced exactly once.

36   *Returns*: An iterator that points to the copy of the first element inserted into `a`, or `p` if `i == j`.

`a.insert(p, il)`   is an expression with the following properties:

37   *Effects*: Equivalent to `a.insert(p, il.begin(), il.end())`.

`a.erase(q)`   is an expression with the following properties:

38   *Result*: A prvalue of type `iterator`.

39   *Preconditions*: For `vector` and `deque`, `T` is *Cpp17MoveAssignable*.

40   *Effects*: Erases the element pointed to by `q`.

41   *Returns*: An iterator that points to the element immediately following `q` prior to the element being erased. If no such element exists, `a.end()` is returned.

`a.erase(q1, q2)`   is an expression with the following properties:

42   *Result*: A prvalue of type `iterator`.

43   *Preconditions*: For `vector` and `deque`, `T` is *Cpp17MoveAssignable*.

44   *Effects*: Erases the elements in the range `[q1, q2)`.

45   *Returns*: An iterator that points to the element pointed to by `q2` prior to any elements being erased. If no such element exists, `a.end()` is returned.

`a.clear()`   is an expression with the following properties:

46   *Result*: `void`

47   *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.

48   *Postconditions*: `a.empty()` is `true`.

49   *Complexity*: Linear.

`a.assign(i, j)`   is an expression with the following properties:

50   *Result*: `void`

51   *Preconditions*: `T` is *Cpp17EmplaceConstructible* into `X` from `*i` and assignable from `*i`. For `vector`, if the iterator does not meet the forward iterator requirements (23.3.5.5), `T` is also *Cpp17MoveInsertable* into `X`. Neither `i` nor `j` are iterators into `a`.

52   *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(il)`   is an expression with the following properties:

53   *Effects*: Equivalent to `a.assign(il.begin(), il.end())`.

`a.assign(n, t)`   is an expression with the following properties:

54   *Result*: `void`

55   *Preconditions*: `T` is *Cpp17CopyInsertable* into `X` and *Cpp17CopyAssignable*. `t` is not a reference into `a`.

56   *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.

57   For every sequence container defined in this Clause and in Clause 21:

(57.1)   — If the constructor

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

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

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

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

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

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

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

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

58 The following operations are provided for some types of sequence containers but not others. An implementation shall implement them so as to take amortized constant time.

`a.front()` is an expression with the following properties:

59 *Result*: `reference`; `const_reference` for constant `a`.

60 *Returns*: `*a.begin()`

61 *Remarks*: Required for `basic_string`, `array`, `deque`, `forward_list`, `list`, and `vector`.

`a.back()` is an expression with the following properties:

62 *Effects*: Equivalent to:

```
auto tmp = a.end();
--tmp;
return *tmp;
```

63 *Remarks*: Required for `basic_string`, `array`, `deque`, `list`, and `vector`.

`a.emplace_front(args)` is an expression with the following properties:

64 *Result*: `reference`

65 *Preconditions*: `T` is *Cpp17EmplaceConstructible* into `X` from `args`.

66 *Effects*: Prepends an object of type `T` constructed with `std::forward<Args>(args)...`.

67 *Returns*: `a.front()`.

68 *Remarks*: Required for `deque`, `forward_list`, and `list`.

`a.emplace_back(args)` is an expression with the following properties:

69 *Result*: `reference`

70 *Preconditions*: `T` is *Cpp17EmplaceConstructible* into `X` from `args`. For `vector`, `T` is also *Cpp17MoveInsertable* into `X`.

71 *Effects*: Appends an object of type `T` constructed with `std::forward<Args>(args)...`.

72 *Returns*: `a.back()`.

73 *Remarks*: Required for `deque`, `list`, and `vector`.

`a.push_front(t)` is an expression with the following properties:

74 *Result*: `void`

75 *Preconditions*: `T` is *Cpp17CopyInsertable* into `X`.

76 *Effects*: Prepends a copy of `t`.

77 *Remarks*: Required for `deque`, `forward_list`, and `list`.

`a.push_front(rv)`   is an expression with the following properties:

78   *Result*: `void`

79   *Preconditions*: `T` is *Cpp17MoveInsertable* into `X`.

80   *Effects*: Prepends a copy of `rv`.

81   *Remarks*: Required for `deque`, `forward_list`, and `list`.

`a.push_back(t)`   is an expression with the following properties:

82   *Result*: `void`

83   *Preconditions*: `T` is *Cpp17CopyInsertable* into `X`.

84   *Effects*: Appends a copy of `t`.

85   *Remarks*: Required for `basic_string`, `deque`, `list`, and `vector`.

`a.push_back(rv)`   is an expression with the following properties:

86   *Result*: `void`

87   *Preconditions*: `T` is *Cpp17MoveInsertable* into `X`.

88   *Effects*: Appends a copy of `rv`.

89   *Remarks*: Required for `basic_string`, `deque`, `list`, and `vector`.

`a.pop_front()`   is an expression with the following properties:

90   *Result*: `void`

91   *Preconditions*: `a.empty()` is `false`.

92   *Effects*: Destroys the first element.

93   *Remarks*: Required for `deque`, `forward_list`, and `list`.

`a.pop_back()`   is an expression with the following properties:

94   *Result*: `void`

95   *Preconditions*: `a.empty()` is `false`.

96   *Effects*: Destroys the last element.

97   *Remarks*: Required for `basic_string`, `deque`, `list`, and `vector`.

`a[n]`   is an expression with the following properties:

98   *Result*: `reference`; `const_reference` for constant `a`

99   *Returns*: `*(a.begin() + n)`

100   *Remarks*: Required for `basic_string`, `array`, `deque`, and `vector`.

`a.at(n)`   is an expression with the following properties:

101   *Result*: `reference`; `const_reference` for constant `a`

102   *Returns*: `*(a.begin() + n)`

103   *Throws*: `out_of_range` if `n >= a.size()`.

104   *Remarks*: Required for `basic_string`, `array`, `deque`, and `vector`.

22.2.5   Node handles   [container.node]

22.2.5.1   Overview   [container.node.overview]

1   A *node handle* is an object that accepts ownership of a single element from an associative container (22.2.7) or an unordered associative container (22.2.8). It may be used to transfer that ownership to another container with compatible nodes. Containers with compatible nodes have the same node handle type. Elements may be transferred in either direction between container types in the same row of Table 73.

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

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

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

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

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

22.2.5.5   Modifiers   [container.node.modifiers]

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

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

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

22.2.6   Insert return type   [container.insert.return]

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

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

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

22.2.7   Associative containers   [associative.reqmts]

22.2.7.1   General   [associative.reqmts.general]

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

2   Each associative container is parameterized on `Key` and an ordering relation `Compare` that induces a strict weak ordering (25.8) on elements of `Key`. In addition, `map` and `multimap` associate an arbitrary *mapped type* `T` with the `Key`. The object of type `Compare` is called the *comparison object* of a container.

3   The phrase "equivalence of keys" means the equivalence relation imposed by the comparison object. That is, two keys `k1` and `k2` are considered to be equivalent if for the comparison object `comp`, `comp(k1, k2) == false && comp(k2, k1) == false`.

    [*Note 1*: This is not necessarily the same as the result of `k1 == k2`. — *end note*]

    For any two keys `k1` and `k2` in the same container, calling `comp(k1, k2)` shall always return the same value.

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

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

6   `iterator` of an associative container is of the bidirectional iterator category. For associative containers where the value type is the same as the key type, both `iterator` and `const_iterator` are constant iterators. It is unspecified whether or not `iterator` and `const_iterator` are the same type.

    [*Note 2*: `iterator` and `const_iterator` have identical semantics in this case, and `iterator` is convertible to `const_-iterator`. Users can avoid violating the one-definition rule by always using `const_iterator` in their function parameter lists. — *end note*]

7   In this subclause,

(7.1)   — `X` denotes an associative container class,

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

(7.3)   — `a2` denotes a value of a type with nodes compatible with type `X` (Table 73),

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

(7.5)   — `u` denotes the name of a variable being declared,

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

(7.7)   — `a_eq` denotes a value of type `X` when `X` supports multiple keys,

(7.8)   — `a_tran` denotes a possibly `const` value of type `X` when the *qualified-id* `X::key_compare::is_transpa-rent` is valid and denotes a type (13.10.3),

(7.9)   — `i` and `j` meet the *Cpp17InputIterator* requirements and refer to elements implicitly convertible to `value_type`,

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

(7.11)   — `p` denotes a valid constant iterator to `a`,

(7.12)   — `q` denotes a valid dereferenceable constant iterator to `a`,

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

(7.14)   — `[q1, q2)` denotes a valid range of constant iterators in `a`,

(7.15)   — `il` designates an object of type `initializer_list<value_type>`,

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

(7.17)   — `k` denotes a value of type `X::key_type`, and

(7.18)   — `c` denotes a possibly `const` value of type `X::key_compare`;

(7.19)   — `kl` is a value such that `a` is partitioned (25.8) with respect to `c(r, kl)`, with `r` the key value of `e` and `e` in `a`;

(7.20)   — `ku` is a value such that `a` is partitioned with respect to `!c(ku, r)`;

(7.21)   — `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)`.

(7.22)   — `A` denotes the storage allocator used by `X`, if any, or `allocator<X::value_type>` otherwise,

(7.23)   — `m` denotes an allocator of a type convertible to `A`, and `nh` denotes a non-const rvalue of type `X::node_-type`.

8   A type `X` meets the *associative container* requirements if `X` meets all the requirements of an allocator-aware container (22.2.2.1) and the following statements and expressions are well-formed and have the specified semantics, except that for `map` and `multimap`, the requirements placed on `value_type` in 22.2.2.5 apply instead to `key_type` and `mapped_type`.

[*Note 3*: For example, in some cases `key_type` and `mapped_type` are required to be *Cpp17CopyAssignable* even though the associated `value_type`, `pair<const key_type, mapped_type>`, is not *Cpp17CopyAssignable*. — *end note*]

(8.1)   — The name `X::key_type` denotes the type `Key`.

(8.2)   — For `map` and `multimap` only, the name `X::mapped_type` denotes `T`.

(8.3)   — For `set` and `multiset` only, the name `X::value_type` denotes `Key`.

(8.4)   — For `map` and `multimap` only, the name `X::value_type` denotes `pair<const Key, T>`.

(8.5)   — *Preconditions*: `X::value_type` is *Cpp17Erasable* from `X`.

(8.6)   — The name `X::key_compare` denotes `Compare`. *Preconditions*: `key_compare` is *Cpp17CopyConstructible*.

(8.7)   — The name `X::value_compare` denotes a binary predicate type. It is the same as `key_compare` for `set` and `multiset`; is an ordering relation on pairs induced by the first component (i.e., `Key`) for `map` and `multimap`.

(8.8)   — The name `X::node_type` denotes a specialization of the *node-handle* class template (22.2.5), such that the public nested types are the same types as the corresponding types in `X`.

X(c)   is an expression with the following properties:

9   *Effects*: Constructs an empty container. Uses a copy of c as a comparison object.

10   *Complexity*: Constant.

X u = X(); and X u; are statements, each with the following properties:

11   *Preconditions*: key_compare meets the *Cpp17DefaultConstructible* requirements.

12   *Effects*: Constructs an empty container. Uses Compare() as a comparison object.

13   *Complexity*: Constant.

X(i, j, c);   is an expression with the following properties:

14   *Preconditions*: value_type is *Cpp17EmplaceConstructible* into X from *i.

15   *Effects*: Constructs an empty container and inserts elements from the range [i, j) into it; uses c as a comparison object.

16   *Complexity*: $N \log N$ in general, where N has the value distance(i, j); linear if [i, j) is sorted with value_comp().

X(i, j)   is an expression with the following properties:

17   *Preconditions*: key_compare meets the *Cpp17DefaultConstructible* requirements. value_type is *Cpp17EmplaceConstructible* into X from *i.

18   *Effects*: Constructs an empty container and inserts elements from the range [i, j) into it; uses Compare() as a comparison object.

19   *Complexity*: $N \log N$ in general, where N has the value distance(i, j); linear if [i, j) is sorted with value_comp().

X(il, c)   is an expression with the following properties:

20   *Effects*: Equivalent to X(il.begin(), il.end(), c).

X(il)   is an expression with the following properties:

21   *Effects*: Equivalent to X(il.begin(), il.end()).

a = il   is an expression with the following properties:

22   *Result*: X&

23   *Preconditions*: value_type is *Cpp17CopyInsertable* into X and *Cpp17CopyAssignable*.

24   *Effects*: Assigns the range [il.begin(), il.end()) into a. All existing elements of a are either assigned to or destroyed.

25   *Complexity*: $N \log N$ in general, where N has the value il.size() + a.size(); linear if [il.begin(), il.end()) is sorted with value_comp().

b.key_comp()   is an expression with the following properties:

26   *Result*: X::key_compare

27   *Returns*: The comparison object out of which b was constructed.

28   *Complexity*: Constant.

b.value_comp()   is an expression with the following properties:

29   *Result*: X::value_compare

30   *Returns*: An object of value_compare constructed out of the comparison object.

31   *Complexity*: Constant.

a_uniq.emplace(args)   is an expression with the following properties:

32   *Result*: pair<iterator, bool>

33   *Preconditions*: value_type is *Cpp17EmplaceConstructible* into X from args.

34   *Effects*: Inserts a `value_type` object `t` constructed with `std::forward<Args>(args)...` if and only if there is no element in the container with key equivalent to the key of `t`.

35   *Returns*: The `bool` component of the returned pair is `true` if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of `t`.

36   *Complexity*: Logarithmic.

`a_eq.emplace(args)`   is an expression with the following properties:

37   *Result*: `iterator`

38   *Preconditions*: `value_type` is *Cpp17EmplaceConstructible* into `X` from `args`.

39   *Effects*: Inserts a `value_type` object `t` constructed with `std::forward<Args>(args)...`. If a range containing elements equivalent to `t` exists in `a_eq`, `t` is inserted at the end of that range.

40   *Returns*: An iterator pointing to the newly inserted element.

41   *Complexity*: Logarithmic.

`a.emplace_hint(p, args)`   is an expression with the following properties:

42   *Result*: `iterator`

43   *Effects*: Equivalent to `a.emplace(std::forward<Args>(args)...)`, except that the element is inserted as close as possible to the position just prior to `p`.

44   *Returns*: An iterator pointing to the element with the key equivalent to the newly inserted element.

45   *Complexity*: Logarithmic in general, but amortized constant if the element is inserted right before `p`.

`a_uniq.insert(t)`   is an expression with the following properties:

46   *Result*: `pair<iterator, bool>`

47   *Preconditions*: If `t` is a non-const rvalue, `value_type` is *Cpp17MoveInsertable* into `X`; otherwise, `value_type` is *Cpp17CopyInsertable* into `X`.

48   *Effects*: Inserts `t` if and only if there is no element in the container with key equivalent to the key of `t`.

49   *Returns*: The `bool` component of the returned pair is `true` if and only if the insertion takes place, and the `iterator` component of the pair points to the element with key equivalent to the key of `t`.

50   *Complexity*: Logarithmic.

`a_eq.insert(t)`   is an expression with the following properties:

51   *Result*: `iterator`

52   *Preconditions*: If `t` is a non-const rvalue, `value_type` is *Cpp17MoveInsertable* into `X`; otherwise, `value_type` is *Cpp17CopyInsertable* into `X`.

53   *Effects*: Inserts `t` and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to `t` exists in `a_eq`, `t` is inserted at the end of that range.

54   *Complexity*: Logarithmic.

`a.insert(p, t)`   is an expression with the following properties:

55   *Result*: `iterator`

56   *Preconditions*: If `t` is a non-const rvalue, `value_type` is *Cpp17MoveInsertable* into `X`; otherwise, `value_type` is *Cpp17CopyInsertable* into `X`.

57   *Effects*: Inserts `t` if and only if there is no element with key equivalent to the key of `t` in containers with unique keys; always inserts `t` in containers with equivalent keys. `t` is inserted as close as possible to the position just prior to `p`.

58   *Returns*: An iterator pointing to the element with key equivalent to the key of `t`.

59   *Complexity*: Logarithmic in general, but amortized constant if `t` is inserted right before `p`.

`a.insert(i, j)`   is an expression with the following properties:

60   *Result*: `void`

61   *Preconditions*: `value_type` is *Cpp17EmplaceConstructible* into `X` from `*i`. Neither `i` nor `j` are iterators into `a`.

62   *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.

63   *Complexity*: $N \log(\texttt{a.size()} + N)$, where N has the value `distance(i, j)`.

`a.insert(il)`   is an expression with the following properties:

64   *Effects*: Equivalent to `a.insert(il.begin(), il.end())`.

`a_uniq.insert(nh)`   is an expression with the following properties:

65   *Result*: `insert_return_type`

66   *Preconditions*: `nh` is empty or `a_uniq.get_allocator() == nh.get_allocator()` is `true`.

67   *Effects*: If `nh` is empty, has no effect. Otherwise, inserts the element owned by `nh` if and only if there is no element in the container with a key equivalent to `nh.key()`.

68   *Returns*: If `nh` is empty, `inserted` is `false`, `position` is `end()`, and `node` is empty. Otherwise if the insertion took place, `inserted` is `true`, `position` points to the inserted element, and `node` is empty; if the insertion failed, `inserted` is `false`, `node` has the previous value of `nh`, and `position` points to an element with a key equivalent to `nh.key()`.

      Complexity: Logarithmic.

`a_eq.insert(nh)`   is an expression with the following properties:

69   *Result*: `iterator`

70   *Preconditions*: `nh` is empty or `a_eq.get_allocator() == nh.get_allocator()` is `true`.

71   *Effects*: If `nh` is empty, has no effect and returns `a_eq.end()`. Otherwise, inserts the element owned by `nh` and returns an iterator pointing to the newly inserted element. If a range containing elements with keys equivalent to `nh.key()` exists in `a_eq`, the element is inserted at the end of that range.

72   *Postconditions*: `nh` is empty.

73   *Complexity*: Logarithmic.

`a.insert(p, nh)`   is an expression with the following properties:

74   *Result*: `iterator`

75   *Preconditions*: `nh` is empty or `a.get_allocator() == nh.get_allocator()` is `true`.

76   *Effects*: If `nh` is empty, has no effect and returns `a.end()`. Otherwise, inserts the element owned by `nh` if and only if there is no element with key equivalent to `nh.key()` in containers with unique keys; always inserts the element owned by `nh` in containers with equivalent keys. The element is inserted as close as possible to the position just prior to `p`.

77   *Postconditions*: `nh` is empty if insertion succeeds, unchanged if insertion fails.

78   *Returns*: An iterator pointing to the element with key equivalent to `nh.key()`.

79   *Complexity*: Logarithmic in general, but amortized constant if the element is inserted right before `p`.

`a.extract(k)`   is an expression with the following properties:

80   *Result*: `node_type`

81   *Effects*: Removes the first element in the container with key equivalent to `k`.

82   *Returns*: A `node_type` owning the element if found, otherwise an empty `node_type`.

83   *Complexity*: $\log(\texttt{a.size()})$

`a.extract(q)`   is an expression with the following properties:

84   *Result*: `node_type`

85   *Effects*: Removes the element pointed to by `q`.

86   *Returns*: A `node_type` owning that element.

87   *Complexity*: Amortized constant.

`a.merge(a2)`   is an expression with the following properties:

88   *Result*: `void`

89   *Preconditions*: `a.get_allocator() == a2.get_allocator()`.

90   *Effects*: Attempts to extract each element in `a2` and insert it into `a` using the comparison object of `a`. In containers with unique keys, if there is an element in `a` with key equivalent to the key of an element from `a2`, then that element is not extracted from `a2`.

91   *Postconditions*: Pointers and references to the transferred elements of `a2` refer to those same elements but as members of `a`. Iterators referring to the transferred elements will continue to refer to their elements, but they now behave as iterators into `a`, not into `a2`.

92   *Throws*: Nothing unless the comparison object throws.

93   *Complexity*: $N \log(\mathtt{a.size()}+N)$, where N has the value `a2.size()`.

`a.erase(k)`   is an expression with the following properties:

94   *Result*: `size_type`

95   *Effects*: Erases all elements in the container with key equivalent to `k`.

96   *Returns*: The number of erased elements.

97   *Complexity*: $\log(\mathtt{a.size()}) + \mathtt{a.count(k)}$

`a.erase(q)`   is an expression with the following properties:

98   *Result*: `iterator`

99   *Effects*: Erases the element pointed to by `q`.

Returns: An iterator pointing to the element immediately following `q` prior to the element being erased. If no such element exists, returns `a.end()`.

100   *Complexity*: Amortized constant.

`a.erase(r)`   is an expression with the following properties:

101   *Result*: `iterator`

102   *Effects*: Erases the element pointed to by `r`.

103   *Returns*: An iterator pointing to the element immediately following `r` prior to the element being erased. If no such element exists, returns `a.end()`.

104   *Complexity*: Amortized constant.

`a.erase(q1, q2)`   is an expression with the following properties:

105   *Result*: `iterator`

106   *Effects*: Erases all the elements in the range `[q1, q2)`.

107   *Returns*: An iterator pointing to the element pointed to by `q2` prior to any elements being erased. If no such element exists, `a.end()` is returned.

108   *Complexity*: $\log(\mathtt{a.size()}) + N$, where N has the value `distance(q1, q2)`.

`a.clear()`   is an expression with the following properties:

109   *Effects*: Equivalent to `a.erase(a.begin(), a.end())`.

110   *Postconditions*: `a.empty()` is `true`.

111   *Complexity*: Linear in `a.size()`.

`b.find(k)`   is an expression with the following properties:

112   *Result*: `iterator`; `const_iterator` for constant `b`.

113   *Returns*: An iterator pointing to an element with the key equivalent to `k`, or `b.end()` if such an element is not found.

114   *Complexity*: Logarithmic.

`a_tran.find(ke)`   is an expression with the following properties:

115   *Result*: `iterator`; `const_iterator` for constant `a_tran`.

116   *Returns*: An iterator pointing to an element with key `r` such that `!c(r, ke) && !c(ke, r)` is `true`, or `a_tran.end()` if such an element is not found.

117   *Complexity*: Logarithmic.

`b.count(k)`   is an expression with the following properties:

118   *Result*: `size_type`

119   *Returns*: The number of elements with key equivalent to `k`.

120   *Complexity*: $\log(\texttt{b.size()}) + \texttt{b.count(k)}$

`a_tran.count(ke)`   is an expression with the following properties:

121   *Result*: `size_type`

122   *Returns*: The number of elements with key `r` such that `!c(r, ke) && !c(ke, r)`.

123   *Complexity*: $\log(\texttt{a\_tran.size()}) + \texttt{a\_tran.count(ke)}$

`b.contains(k)`   is an expression with the following properties:

124   *Result*: `bool`

125   *Effects*: Equivalent to: `return b.find(k) != b.end();`

`a_tran.contains(ke)`   is an expression with the following properties:

126   *Result*: `bool`

127   *Effects*: Equivalent to: `return a_tran.find(ke) != a_tran.end();`

`b.lower_bound(k)`   is an expression with the following properties:

128   *Result*: `iterator`; `const_iterator` for constant `b`.

129   *Returns*: An iterator pointing to the first element with key not less than `k`, or `b.end()` if such an element is not found.

130   *Complexity*: Logarithmic.

`a_tran.lower_bound(kl)`   is an expression with the following properties:

131   *Result*: `iterator`; `const_iterator` for constant `a_tran`.

132   *Returns*: An iterator pointing to the first element with key `r` such that `!c(r, kl)`, or `a_tran.end()` if such an element is not found.

133   *Complexity*: Logarithmic.

`b.upper_bound(k)`   is an expression with the following properties:

134   *Result*: `iterator`; `const_iterator` for constant `b`.

135   *Returns*: An iterator pointing to the first element with key greater than `k`, or `b.end()` if such an element is not found.

136   *Complexity*: Logarithmic,

`a_tran.upper_bound(ku)`   is an expression with the following properties:

137   *Result*: `iterator`; `const_iterator` for constant `a_tran`.

138   *Returns*: An iterator pointing to the first element with key `r` such that `c(ku, r)`, or `a_tran.end()` if such an element is not found.

139   *Complexity*: Logarithmic.

`b.equal_range(k)`   is an expression with the following properties:

140   *Result*: `pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for constant `b`.

141   *Effects*: Equivalent to: `return make_pair(b.lower_bound(k), b.upper_bound(k));`

142   *Complexity*: Logarithmic.

`a_tran.equal_range(ke)`   is an expression with the following properties:

143   *Result*: `pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for constant `a_tran`.

144   *Effects*: Equivalent to: `return make_pair(a_tran.lower_bound(ke), a_tran.upper_bound(ke));`

145   *Complexity*: Logarithmic.

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

147   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.

148   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
```

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

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

150   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.

151   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 (13.10.3).

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

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

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

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

22.2.7.2   Exception safety guarantees   [associative.reqmts.except]

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

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

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

22.2.8   Unordered associative containers   [unord.req]

22.2.8.1   General   [unord.req.general]

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

2   Unordered associative containers conform to the requirements for Containers (22.2), except that the expressions `a == b` and `a != b` have different semantics than for the other container types.

3   Each unordered associative container is parameterized by `Key`, by a function object type `Hash` that meets the *Cpp17Hash* requirements (16.4.4.5) and acts as a hash function for argument values of type `Key`, and by a binary predicate `Pred` that induces an equivalence relation on values of type `Key`. Additionally, `unordered_map` and `unordered_multimap` associate an arbitrary *mapped type* T with the `Key`.

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

5   Two values `k1` and `k2` are considered equivalent if the container's key equality predicate `pred(k1, k2)` is valid and returns `true` when passed those values. If `k1` and `k2` are equivalent, the container's hash function shall return the same value for both.

[*Note 1*: Thus, when an unordered associative container is instantiated with a non-default `Pred` parameter it usually needs a non-default `Hash` parameter as well. — *end note*]

For any two keys `k1` and `k2` in the same container, calling `pred(k1, k2)` shall always return the same value. For any key `k` in a container, calling `hash(k)` shall always return the same value.

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

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

8   For unordered containers where the value type is the same as the key type, both `iterator` and `const_iterator` are constant iterators. It is unspecified whether or not `iterator` and `const_iterator` are the same type.

[*Note 2*: `iterator` and `const_iterator` have identical semantics in this case, and `iterator` is convertible to `const_iterator`. Users can avoid violating the one-definition rule by always using `const_iterator` in their function parameter lists. — *end note*]

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

10   In this subclause,

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

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

(10.3)   — `a2` denotes a value of a type with nodes compatible with type `X` (Table 73),

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

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

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

(10.7)   — `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 (13.10.3),

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

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

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

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

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

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

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

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

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

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

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

(10.19) — `ke` is a value such that

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

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

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

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

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

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

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

11 A type `X` meets the *unordered associative container* requirements if `X` meets all the requirements of an allocator-aware container (22.2.2.1) and the following statements and expressions are well-formed and have the specified semantics, except that for `unordered_map` and `unordered_multimap`, the requirements placed on `value_type` in 22.2.2.5 apply instead to `key_type` and `mapped_type`.

[*Note 3*: For example, `key_type` and `mapped_type` are sometimes required to be *Cpp17CopyAssignable* even though the associated `value_type`, `pair<const key_type, mapped_type>`, is not *Cpp17CopyAssignable*. — *end note*]

(11.1) — The name `X::key_type` denotes the type `Key`.

(11.2) — For `unordered_map` and `unordered_multimap` only, the name `X::mapped_type` denotes `T`.

(11.3) — For `unordered_set` and `unordered_multiset` only, the name `X::value_type` denotes `Key`.

(11.4) — For `unordered_map` and `unordered_multimap` only, the name `X::value_type` denotes `pair<const Key, T>`.

(11.5) — *Preconditions*: `value_type` is *Cpp17Erasable* from `X`.

(11.6) — The name `X::hasher` denotes the type `Hash`. *Preconditions*: `Hash` is a unary function object type such that the expression `hf(k)` has type `size_t`.

(11.7) — The name `X::key_equal` denotes the type `Pred`. *Preconditions*: `Pred` meets the *Cpp17CopyConstructible* requirements. `Pred` is a binary predicate that takes two arguments of type `Key`. `Pred` is an equivalence relation.

(11.8) — The name `X::local_iterator` denotes an iterator type whose category, value type, difference type, and pointer and reference types are the same as `X::iterator`'s.

[*Note 4*: A `local_iterator` object can be used to iterate through a single bucket, but cannot be used to iterate across buckets. — *end note*]

(11.9) — The name `X::const_local_iterator` denotes an iterator type whose category, value type, difference type, and pointer and reference types are the same as `X::const_iterator`'s. A `const_local_iterator` object may be used to iterate through a single bucket, but may not be used to iterate across buckets.

(11.10) — The name `X::node_type` denotes a specialization of a *node-handle* class template (22.2.5), such that the public nested types are the same types as the corresponding types in `X`.

`X(n, hf, eq)`   is an expression with the following properties:

12 *Effects*: Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `eq` as the key equality predicate.

13 *Complexity*: $\mathscr{O}(n)$

`X(n, hf)`   is an expression with the following properties:

14 *Preconditions*: `key_equal` meets the *Cpp17DefaultConstructible* requirements.

15   *Effects*: Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `key_equal()` as the key equality predicate.

16   *Complexity*: $\mathscr{O}(\texttt{n})$

`X(n)`   is an expression with the following properties:

17   *Preconditions*: `hasher` and `key_equal` meet the *Cpp17DefaultConstructible* requirements.

18   *Effects*: Constructs an empty container with at least `n` buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate.

19   *Complexity*: $\mathscr{O}(\texttt{n})$

The statements `X a = X()` and `X a;` each satisfy the following properties:

     Preconditions: `hasher` and `key_equal` meet the *Cpp17DefaultConstructible* requirements.

20   *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.

21   *Complexity*: Constant.

`X(i, j, n, hf, eq)`   is an expression with the following properties:

22   *Preconditions*: `value_type` is *Cpp17EmplaceConstructible* into X from `*i`.

23   *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.

24   *Complexity*: Average case $\mathscr{O}(N)$ (N is `distance(i, j)`), worst case $\mathscr{O}(N^2)$.

`X(i, j, n, hf)`   is an expression with the following properties:

25   *Preconditions*: `key_equal` meets the *Cpp17DefaultConstructible* requirements. `value_type` is *Cpp17-EmplaceConstructible* into X from `*i`.

26   *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.

27   *Complexity*: Average case $\mathscr{O}(N)$ (N is `distance(i, j)`), worst case $\mathscr{O}(N^2)$.

`X(i, j, n)`   is an expression with the following properties:

28   *Preconditions*: `hasher` and `key_equal` meet the *Cpp17DefaultConstructible* requirements. `value_type` is *Cpp17EmplaceConstructible* into X from `*i`.

29   *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.

30   *Complexity*: Average case $\mathscr{O}(N)$ (N is `distance(i, j)`), worst case $\mathscr{O}(N^2)$.

`X(i, j)`   is an expression with the following properties:

31   *Preconditions*: `hasher` and `key_equal` meet the *Cpp17DefaultConstructible* requirements. `value_type` is *Cpp17EmplaceConstructible* into X from `*i`.

32   *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.

33   *Complexity*: Average case $\mathscr{O}(N)$ (N is `distance(i, j)`), worst case $\mathscr{O}(N^2)$.

`X(il)`   is an expression with the following properties:

34   *Effects*: Equivalent to `X(il.begin(), il.end())`.

`X(il, n)`   is an expression with the following properties:

35   *Effects*: Equivalent to `X(il.begin(), il.end(), n)`.

`X(il, n, hf)`   is an expression with the following properties:

36   *Effects*: Equivalent to `X(il.begin(), il.end(), n, hf)`.

X(il, n, hf, eq)   is an expression with the following properties:

37   *Effects*: Equivalent to X(il.begin(), il.end(), n, hf, eq).

X(b)   is an expression with the following properties:

38   *Effects*: In addition to the container requirements (22.2.2.1), copies the hash function, predicate, and maximum load factor.

39   *Complexity*: Average case linear in b.size(), worst case quadratic.

a = b   is an expression with the following properties:

40   *Result*: X&

41   *Effects*: In addition to the container requirements, copies the hash function, predicate, and maximum load factor.

42   *Complexity*: Average case linear in b.size(), worst case quadratic.

a = il   is an expression with the following properties:

43   *Result*: X&

44   *Preconditions*: value_type is *Cpp17CopyInsertable* into X and *Cpp17CopyAssignable*.

45   *Effects*: Assigns the range [il.begin(), il.end()) into a. All existing elements of a are either assigned to or destroyed.

46   *Complexity*: Average case linear in il.size(), worst case quadratic.

b.hash_function()   is an expression with the following properties:

47   *Result*: hasher

48   *Returns*: b's hash function.

49   *Complexity*: Constant.

b.key_eq()   is an expression with the following properties:

50   *Result*: key_equal

51   *Returns*: b's key equality predicate.

52   *Complexity*: Constant.

a_uniq.emplace(args)   is an expression with the following properties:

53   *Result*: pair<iterator, bool>

54   *Preconditions*: value_type is *Cpp17EmplaceConstructible* into X from args.

55   *Effects*: Inserts a value_type object t constructed with std::forward<Args>(args)... if and only if there is no element in the container with key equivalent to the key of t.

56   *Returns*: The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of t.

57   *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(\text{a\_uniq.size()})$.

a_eq.emplace(args)   is an expression with the following properties:

58   *Result*: iterator

59   *Preconditions*: value_type is *Cpp17EmplaceConstructible* into X from args.

60   *Effects*: Inserts a value_type object t constructed with std::forward<Args>(args)... and

61   *Returns*: An iterator pointing to the newly inserted element.

62   *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(\text{a\_eq.size()})$.

a.emplace_hint(p, args)   is an expression with the following properties:

63   *Result*: iterator

64   *Preconditions*: value_type is *Cpp17EmplaceConstructible* into X from args.

65   *Effects*: Equivalent to `a.emplace(std::forward<Args>(args)...)`.

66   *Returns*: An iterator pointing to the element with the key equivalent to the newly inserted element. The `const_iterator p` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.

67   *Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{a.size()})$.

`a_uniq.insert(t)`   is an expression with the following properties:

68   *Result*: `pair<iterator, bool>`

69   *Preconditions*: If `t` is a non-const rvalue, `value_type` is *Cpp17MoveInsertable* into `X`; otherwise, `value_type` is *Cpp17CopyInsertable* into `X`.

70   *Effects*: Inserts `t` if and only if there is no element in the container with key equivalent to the key of `t`.

71   *Returns*: 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`.

72   *Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{a\_uniq.size()})$.

`a_eq.insert(t)`   is an expression with the following properties:

73   *Result*: `iterator`

74   *Preconditions*: If `t` is a non-const rvalue, `value_type` is *Cpp17MoveInsertable* into `X`; otherwise, `value_type` is *Cpp17CopyInsertable* into `X`.

75   *Effects*: Inserts `t`.

76   *Returns*: An iterator pointing to the newly inserted element.

77   *Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{a\_eq.size()})$.

`a.insert(p, t)`   is an expression with the following properties:

78   *Result*: `iterator`

79   *Preconditions*: If `t` is a non-const rvalue, `value_type` is *Cpp17MoveInsertable* into `X`; otherwise, `value_type` is *Cpp17CopyInsertable* into `X`.

80   *Effects*: Equivalent to `a.insert(t)`. The iterator `p` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.

81   *Returns*: An iterator pointing to the element with the key equivalent to that of `t`.

82   *Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{a.size()})$.

`a.insert(i, j)`   is an expression with the following properties:

83   *Result*: `void`

84   *Preconditions*: `value_type` is *Cpp17EmplaceConstructible* into `X` from `*i`. Neither `i` nor `j` are iterators into `a`.

85   *Effects*: Equivalent to `a.insert(t)` for each element in `[i,j)`.

86   *Complexity*: Average case $\mathscr{O}(N)$, where N is `distance(i, j)`, worst case $\mathscr{O}(N(\texttt{a.size()} + 1))$.

`a.insert(il)`   is an expression with the following properties:

87   *Effects*: Equivalent to `a.insert(il.begin(), il.end())`.

`a_uniq.insert(nh)`   is an expression with the following properties:

88   *Result*: `insert_return_type`

89   *Preconditions*: `nh` is empty or `a_uniq.get_allocator() == nh.get_allocator()` is `true`.

90   *Effects*: If `nh` is empty, has no effect. Otherwise, inserts the element owned by `nh` if and only if there is no element in the container with a key equivalent to `nh.key()`.

91   *Postconditions*: If `nh` is empty, `inserted` is `false`, `position` is `end()`, and `node` is empty. Otherwise if the insertion took place, `inserted` is `true`, `position` points to the inserted element, and `node` is empty; if the insertion failed, `inserted` is `false`, `node` has the previous value of `nh`, and `position` points to an element with a key equivalent to `nh.key()`.

92   *Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{a\_uniq.size()})$.

`a_eq.insert(nh)`   is an expression with the following properties:

93   *Result*: `iterator`

94   *Preconditions*: `nh` is empty or `a_eq.get_allocator() == nh.get_allocator()` is `true`.

95   *Effects*: If `nh` is empty, has no effect and returns `a_eq.end()`. Otherwise, inserts the element owned by `nh` and returns an iterator pointing to the newly inserted element.

96   *Postconditions*: `nh` is empty.

97   *Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{a\_eq.size()})$.

`a.insert(q, nh)`   is an expression with the following properties:

98   *Result*: `iterator`

99   *Preconditions*: `nh` is empty or `a.get_allocator() == nh.get_allocator()` is `true`.

100   *Effects*: If `nh` is empty, has no effect and returns `a.end()`. Otherwise, inserts the element owned by `nh` if and only if there is no element with key equivalent to `nh.key()` in containers with unique keys; always inserts the element owned by `nh` in containers with equivalent keys. The iterator `q` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.

      Postconditions: `nh` is empty if insertion succeeds, unchanged if insertion fails.

101   *Returns*: An iterator pointing to the element with key equivalent to `nh.key()`.

102   *Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{a.size()})$.

`a.extract(k)`   is an expression with the following properties:

103   *Result*: `node_type`

104   *Effects*: Removes an element in the container with key equivalent to `k`.

105   *Returns*: A `node_type` owning the element if found, otherwise an empty `node_type`.

106   *Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{a.size()})$.

`a.extract(q)`   is an expression with the following properties:

107   *Result*: `node_type`

108   *Effects*: Removes the element pointed to by `q`.

      Returns: A `node_type` owning that element.

109   *Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{a.size()})$.

`a.merge(a2)`   is an expression with the following properties:

110   *Result*: `void`

111   *Preconditions*: `a.get_allocator() == a2.get_allocator()`.

112   *Effects*: Attempts to extract each element in `a2` and insert it into `a` using the hash function and key equality predicate of `a`. In containers with unique keys, if there is an element in `a` with key equivalent to the key of an element from `a2`, then that element is not extracted from `a2`.

113   *Postconditions*: Pointers and references to the transferred elements of `a2` refer to those same elements but as members of `a`. Iterators referring to the transferred elements and all iterators referring to `a` will be invalidated, but iterators to elements remaining in `a2` will remain valid.

114   *Complexity*: Average case $\mathscr{O}(N)$, where N is `a2.size()`, worst case $\mathscr{O}(N*\texttt{a.size()} + N)$.

`a.erase(k)`   is an expression with the following properties:

115   *Result*: `size_type`

116   *Effects*: Erases all elements with key equivalent to `k`.

117   *Returns*: The number of elements erased.

118   *Complexity*: Average case $\mathscr{O}(\texttt{a.count(k)})$, worst case $\mathscr{O}(\texttt{a.size()})$.

a.erase(q)　　is an expression with the following properties:

119　　　*Result*: iterator

120　　　*Effects*: Erases the element pointed to by q.

121　　　*Returns*: The iterator immediately following q prior to the erasure.

122　　　*Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{a.size()})$.

a.erase(r)　　is an expression with the following properties:

123　　　*Result*: iterator

124　　　*Effects*: Erases the element pointed to by r.

125　　　*Returns*: The iterator immediately following r prior to the erasure.

126　　　*Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{a.size()})$.

a.erase(q1, q2)　　is an expression with the following properties:

127　　　*Result*: iterator

128　　　*Effects*: Erases all elements in the range [q1, q2).

129　　　*Returns*: The iterator immediately following the erased elements prior to the erasure.

130　　　*Complexity*: Average case linear in distance(q1, q2), worst case $\mathscr{O}(\texttt{a.size()})$.

a.clear()　　is an expression with the following properties:

131　　　*Result*: void

132　　　*Effects*: Erases all elements in the container.

133　　　*Postconditions*: a.empty() is true.

134　　　*Complexity*: Linear in a.size().

b.find(k)　　is an expression with the following properties:

135　　　*Result*: iterator; const_iterator for const b.

136　　　*Returns*: An iterator pointing to an element with key equivalent to k, or b.end() if no such element
　　　exists.

137　　　*Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{b.size()})$.

a_tran.find(ke)　　is an expression with the following properties:

138　　　*Result*: iterator; const_iterator for const a_tran.

139　　　*Returns*: An iterator pointing to an element with key equivalent to ke, or a_tran.end() if no such
　　　element exists.

140　　　*Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}(\texttt{a\_tran.size()})$.

b.count(k)　　is an expression with the following properties:

141　　　*Result*: size_type

142　　　*Returns*: The number of elements with key equivalent to k.

143　　　*Complexity*: Average case $\mathscr{O}(\texttt{b.count(k)})$, worst case $\mathscr{O}(\texttt{b.size()})$.

a_tran.count(ke)　　is an expression with the following properties:

144　　　*Result*: size_type

145　　　*Returns*: The number of elements with key equivalent to ke.

146　　　*Complexity*: Average case $\mathscr{O}(\texttt{a\_tran.count(ke)})$, worst case $\mathscr{O}(\texttt{a\_tran.size()})$.

b.contains(k)　　is an expression with the following properties:

147　　　*Effects*: Equivalent to b.find(k) != b.end().

`a_tran.contains(ke)`   is an expression with the following properties:

148   *Effects*: Equivalent to `a_tran.find(ke) != a_tran.end()`.

`b.equal_range(k)`   is an expression with the following properties:

149   *Result*: `pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for const `b`.

150   *Returns*: A range containing all elements with keys equivalent to `k`. Returns `make_pair(b.end(), b.end())` if no such elements exist.

151   *Complexity*: Average case $\mathscr{O}(\texttt{b.count(k)})$, worst case $\mathscr{O}(\texttt{b.size()})$.

`a_tran.equal_range(ke)`   is an expression with the following properties:

152   *Result*: `pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for const `a_tran`.

153   *Returns*: A range containing all elements with keys equivalent to `ke`. Returns `make_pair(a_tran.end(), a_tran.end())` if no such elements exist.

154   *Complexity*: Average case $\mathscr{O}(\texttt{a\_tran.count(ke)})$, worst case $\mathscr{O}(\texttt{a\_tran.size()})$.

`b.bucket_count()`   is an expression with the following properties:

155   *Result*: `size_type`

156   *Returns*: The number of buckets that `b` contains.

157   *Complexity*: Constant.

`b.max_bucket_count()`   is an expression with the following properties:

158   *Result*: `size_type`

159   *Returns*: An upper bound on the number of buckets that `b` can ever contain.

160   *Complexity*: Constant.

`b.bucket(k)`   is an expression with the following properties:

161   *Result*: `size_type`

162   *Preconditions*: `b.bucket_count() > 0`.

163   *Returns*: The index of the bucket in which elements with keys equivalent to `k` would be found, if any such element existed. The return value is in the range `[0, b.bucket_count())`.

164   *Complexity*: Constant.

`b.bucket_size(n)`   is an expression with the following properties:

165   *Result*: `size_type`

166   *Preconditions*: `n` shall be in the range `[0, b.bucket_count())`.

167   *Returns*: The number of elements in the nth bucket.

168   *Complexity*: $\mathscr{O}(\texttt{b.bucket\_size(n)})$

`b.begin(n)`   is an expression with the following properties:

169   *Result*: `local_iterator`; `const_local_iterator` for const `b`.

170   *Preconditions*: `n` is in the range `[0, b.bucket_count())`.

171   *Returns*: An iterator referring to the first element in the bucket. If the bucket is empty, then `b.begin(n) == b.end(n)`.

172   *Complexity*: Constant.

`b.end(n)`   is an expression with the following properties:

173   *Result*: `local_iterator`; `const_local_iterator` for const `b`.

174   *Preconditions*: `n` is in the range `[0, b.bucket_count())`.

175   *Returns*: An iterator which is the past-the-end value for the bucket.

176   *Complexity*: Constant.

`b.cbegin(n)`   is an expression with the following properties:

177   *Result*: `const_local_iterator`

178   *Preconditions*: `n` shall be in the range `[0, b.bucket_count())`.

179   *Returns*: An iterator referring to the first element in the bucket. If the bucket is empty, then `b.cbegin(n) == b.cend(n)`.

180   *Complexity*: Constant.

`b.cend(n)`   is an expression with the following properties:

181   *Result*: `const_local_iterator`

182   *Preconditions*: `n` is in the range `[0, b.bucket_count())`.

183   *Returns*: An iterator which is the past-the-end value for the bucket.

184   *Complexity*: Constant.

`b.load_factor()`   is an expression with the following properties:

185   *Result*: `float`

186   *Returns*: The average number of elements per bucket.

187   *Complexity*: Constant.

`b.max_load_factor()`   is an expression with the following properties:

188   *Result*: `float`

189   *Returns*: A positive number that the container attempts to keep the load factor less than or equal to. The container automatically increases the number of buckets as necessary to keep the load factor below this number.

190   *Complexity*: Constant.

`a.max_load_factor(z)`   is an expression with the following properties:

191   *Result*: `void`

192   *Preconditions*: `z` is positive. May change the container's maximum load factor, using `z` as a hint.

193   *Complexity*: Constant.

`a.rehash(n)`   is an expression with the following properties:

194   *Result*: `void`

195   *Postconditions*: `a.bucket_count() >= a.size() / a.max_load_factor()` and `a.bucket_count() >= n`.

196   *Complexity*: Average case linear in `a.size()`, worst case quadratic.

`a.reserve(n)`   is an expression with the following properties:

197   *Effects*: Equivalent to `a.rehash(ceil(n / a.max_load_factor()))`.

198   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 ith 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 $\mathscr{O}(N^2)$, 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[218] of the partition into equivalent-key groups produced by `Pred`.

199 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.

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

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

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

203 The member function templates `find`, `count`, `equal_range`, and `contains` shall not participate in overload resolution unless the *qualified-id*s `Pred::is_transparent` and `Hash::is_transparent` are both valid and denote types (13.10.3).

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

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

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

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

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

22.2.8.2 Exception safety guarantees [unord.req.except]

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

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

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

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

22.3 Sequence containers [sequences]

22.3.1 In general [sequences.general]

1 The headers `<array>` (22.3.2), `<deque>` (22.3.3), `<forward_list>` (22.3.4), `<list>` (22.3.5), and `<vector>` (22.3.6) define class templates that meet the requirements for sequence containers.

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

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

22.3.2 Header `<array>` synopsis [array.syn]

```
#include <compare>            // see 17.11.1
#include <initializer_list>   // see 17.10.2
```

---

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

30   Regular expressions library [re]

30.1   General [re.general]

¹ This Clause describes components that C++ programs may use to perform operations involving regular expression matching and searching.

² The following subclauses describe a basic regular expression class template and its traits that can handle char-like (21.1) template arguments, two specializations of this class template that handle sequences of `char` and `wchar_t`, a class template that holds the result of a regular expression match, a series of algorithms that allow a character sequence to be operated upon by a regular expression, and two iterator types for enumerating regular expression matches, as summarized in Table 126.

Table 126: Regular expressions library summary   [tab:re.summary]

	Subclause	Header
30.2	Requirements	
30.4	Constants	`<regex>`
30.5	Exception type	
30.6	Traits	
30.7	Regular expression template	
30.8	Submatches	
30.9	Match results	
30.10	Algorithms	
30.11	Iterators	
30.12	Grammar	

30.2   Requirements [re.req]

¹ This subclause defines requirements on classes representing regular expression traits.

[*Note 1*: The class template `regex_traits`, defined in 30.6, meets these requirements. — *end note*]

² The class template `basic_regex`, defined in 30.7, needs a set of related types and functions to complete the definition of its semantics. These types and functions are provided as a set of member *typedef-name*s and functions in the template parameter `traits` used by the `basic_regex` class template. This subclause defines the semantics of these members.

³ To specialize class template `basic_regex` for a character container `CharT` and its related regular expression traits class `Traits`, use `basic_regex<CharT, Traits>`.

⁴ In the following requirements,

(4.1)   — `X` denotes a traits class defining types and functions for the character container type `charT`;

(4.2)   — `u` is an object of type `X`;

(4.3)   — `v` is an object of type `const X`;

(4.4)   — `p` is a value of type `const charT*`;

(4.5)   — `I1` and `I2` are input iterators (23.3.5.3);

(4.6)   — `F1` and `F2` are forward iterators (23.3.5.5);

(4.7)   — `c` is a value of type `const charT`;

(4.8)   — `s` is an object of type `X::string_type`;

(4.9)   — `cs` is an object of type `const X::string_type`;

(4.10)   — `b` is a value of type `bool`;

(4.11)   — `I` is a value of type `int`;

(4.12)   — `cl` is an object of type `X::char_class_type`; and

(4.13)　　— `loc` is an object of type `X::locale_type`.

5　A traits class `X` meets the regular expression traits requirements if the following types and expressions are well-formed and have the specified semantics.

(5.1)　　— The type `X::char_type` denotes `charT`, the character container type used in the implementation of class template `basic_regex`.

(5.2)　　— The type `X::string_type` denotes the type `basic_string<charT>`.

(5.3)　　— The type `X::locale_type` denotes a copy constructible type that represents the locale used by the traits class.

(5.4)　　— The type `X::char_class_type` denotes a bitmask type (16.3.3.3.4) representing a particular character classification.

`X::length(p)`

6　*Return type*: `size_t`

7　*Returns*: The smallest `i` such that `p[i] == 0`.

8　*Complexity*: Linear in `i`.

`v.translate(c)`

9　*Return type*: `X::char_type`

10　*Returns*: A character such that for any character `d` that is to be considered equivalent to `c` then `v.translate(c) == v.translate(d)`.

`v.translate_nocase(c)`

11　*Return type*: `X::char_type`

12　*Returns*: For all characters `C` that are to be considered equivalent to `c` when comparisons are to be performed without regard to case, then `v.translate_nocase(c) == v.translate_nocase(C)`.

`v.transform(F1, F2)`

13　*Return type*: `X::string_type`

14　*Returns*: A sort key for the character sequence designated by the iterator range `[F1, F2)` such that if the character sequence `[G1, G2)` sorts before the character sequence `[H1, H2)` then `v.transform(G1, G2) < v.transform(H1, H2)`.

`v.transform_primary(F1, F2)`

15　*Return type*: `X::string_type`

16　*Returns*: A sort key for the character sequence designated by the iterator range `[F1, F2)` such that if the character sequence `[G1, G2)` sorts before the character sequence `[H1, H2)` when character case is not considered then `v.transform_primary(G1, G2) < v.transform_primary(H1, H2)`.

`v.lookup_collatename(F1, F2)`

17　*Return type*: `X::string_type`

18　*Returns*: A sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range `[F1, F2)`. Returns an empty string if the character sequence is not a valid collating element.

`v.lookup_classname(F1, F2, b)`

19　*Return type*: `X::char_class_type`

20　*Returns*: Converts the character sequence designated by the iterator range `[F1, F2)` into a value of a bitmask type that can subsequently be passed to `isctype`. Values returned from `lookup_classname` can be bitwise OR'ed together; the resulting value represents membership in either of the corresponding character classes. If `b` is `true`, the returned bitmask is suitable for matching characters without regard to their case. Returns `0` if the character sequence is not the name of a character class recognized by `X`. The value returned shall be independent of the case of the characters in the sequence.

`v.isctype(c, cl)`

21   *Return type*: `bool`

22   *Returns*: Returns `true` if character `c` is a member of one of the character classes designated by `cl`, `false` otherwise.

`v.value(c, I)`

23   *Return type*: `int`

24   *Returns*: Returns the value represented by the digit c in base I if the character c is a valid digit in base I; otherwise returns `-1`.

[*Note 2*: The value of I will only be 8, 10, or 16. — *end note*]

`u.imbue(loc)`

25   *Return type*: `X::locale_type`

26   *Effects*: Imbues `u` with the locale `loc` and returns the previous locale used by `u` if any.

`v.getloc()`

27   *Return type*: `X::locale_type`

28   *Returns*: Returns the current locale used by `v`, if any.

29 [*Note 3*: Class template `regex_traits` meets the requirements for a regular expression traits class when it is specialized for `char` or `wchar_t`. This class template is described in the header `<regex>`, and is described in 30.6. — *end note*]

30.3   Header `<regex>` synopsis   [re.syn]

```
#include <compare>            // see 17.11.1
#include <initializer_list>   // see 17.10.2

namespace std {
  // 30.4, regex constants
  namespace regex_constants {
    using syntax_option_type = T1;
    using match_flag_type = T2;
    using error_type = T3;
  }

  // 30.5, class regex_error
  class regex_error;

  // 30.6, class template regex_traits
  template<class charT> struct regex_traits;

  // 30.7, class template basic_regex
  template<class charT, class traits = regex_traits<charT>> class basic_regex;

  using regex  = basic_regex<char>;
  using wregex = basic_regex<wchar_t>;

  // 30.7.6, basic_regex swap
  template<class charT, class traits>
    void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2);

  // 30.8, class template sub_match
  template<class BidirectionalIterator>
    class sub_match;

  using csub_match  = sub_match<const char*>;
  using wcsub_match = sub_match<const wchar_t*>;
  using ssub_match  = sub_match<string::const_iterator>;
  using wssub_match = sub_match<wstring::const_iterator>;
```