ISO/IEC JTC1 SC22 WG21 P2416R2
Author: Jens Maurer
Target audience: LWG
2021-02-14

P2416R2: 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.

These changes were applied to the Working Draft just prior to the Feb 2022 meeting.

Changes vs. R0

- LWG feedback: Presented member typedef requirements with `typename`, but otherwise the same as member function requirements.
- Add "Result" to [structure.specification].

Changes vs. R1

- LWG feedback: Respecify "Result" element and omit "A prvalue of type" from the detailed descriptions.
- LWG feedback: Restore result type of `a.back()`.
- LWG feedback: Use "Result", not "Return type" in [re] specification.

Acknowledgements

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

(3.1) — *Constraints*: the conditions for the function's participation in overload resolution (12.2).

[*Note 1*: Failure to meet such a condition results in the function's silent non-viability. — *end note*]

[*Example 1*: An implementation can express such a condition via a *constraint-expression* (13.5.3). — *end example*]

(3.2) — *Mandates*: the conditions that, if not met, render the program ill-formed.

[*Example 2*: An implementation can express such a condition via the *constant-expression* in a *static_assert-declaration* (9.1). If the diagnostic is to be emitted only after the function has been selected by overload resolution, an implementation can express such a condition via a *constraint-expression* (13.5.3) and also define the function as deleted. — *end example*]

(3.3) — *Preconditions*: the conditions that the function assumes to hold whenever it is called; violation of any preconditions results in undefined behavior.

(3.4) — *Effects*: the actions performed by the function.

(3.5) — *Synchronization*: the synchronization operations (6.9.2) applicable to the function.

(3.6) — *Postconditions*: the conditions (sometimes termed observable results) established by the function.

(3.7) — *Result*: for a *typename-specifier*, a description of the named type; for an *expression*, a description of the type of the expression; the expression is an lvalue if the type is a reference type and a prvalue otherwise.

(3.8) — *Returns*: a description of the value(s) returned by the function.

(3.9) — *Throws*: any exceptions thrown by the function, and the conditions that would cause the exception.

(3.10) — *Complexity*: the time and/or space complexity of the function.

(3.11) — *Remarks*: additional semantic constraints on the function.

(3.12) — *Error conditions*: the error conditions for error codes reported by the function.

4 Whenever the *Effects* element specifies that the semantics of some function `F` are *Equivalent to* some code sequence, then the various elements are interpreted as follows. If `F`'s semantics specifies any *Constraints* or *Mandates* elements, then those requirements are logically imposed prior to the *equivalent-to* semantics. Next, the semantics of the code sequence are determined by the *Constraints*, *Mandates*, *Preconditions*, *Effects*, *Synchronization*, *Postconditions*, *Returns*, *Throws*, *Complexity*, *Remarks*, and *Error conditions* specified for the function invocations contained in the code sequence. The value returned from `F` is specified by `F`'s *Returns* element, or if `F` has no *Returns* element, a non-`void` return from `F` is specified by the `return` statements (8.7.4) in the code sequence. If `F`'s semantics contains a *Throws*, *Postconditions*, or *Complexity* element, then that supersedes any occurrences of that element in the code sequence.

5 For non-reserved replacement and handler functions, Clause 17 specifies two behaviors for the functions in question: their required and default behavior. The *default behavior* describes a function definition provided by the implementation. The *required behavior* describes the semantics of a function definition provided by either the implementation or a C++ program. Where no distinction is explicitly made in the description, the behavior described is the required behavior.

6 If the formulation of a complexity requirement calls for a negative number of operations, the actual requirement is zero operations.[147]

7 Complexity requirements specified in the library clauses are upper bounds, and implementations that provide better complexity guarantees meet the requirements.

8 Error conditions specify conditions where a function may fail. The conditions are listed, together with a suitable explanation, as the `enum class errc` constants (19.5).

16.3.2.5 C library [structure.see.also]

1 Paragraphs labeled "SEE ALSO" contain cross-references to the relevant portions of other standards (Clause 2).

16.3.3 Other conventions [conventions]

16.3.3.1 General [conventions.general]

1 Subclause 16.3.3 describes several editorial conventions used to describe the contents of the C++ standard library. These conventions are for describing implementation-defined types (16.3.3.3), and member functions (16.3.3.4).

147) This simplifies the presentation of complexity requirements in some cases.

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 76.

Table 76: 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 Allocator-aware containers (22.2.2.5) other than `basic_string` construct elements using the function `allocator_traits<allocator_type>::rebind_traits<U>::construct` and destroy elements 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, statements, and expressions are well-formed and have the specified semantics.

```
typename X::value_type
```

2 *Result*: `T`

3 *Preconditions*: `T` is *Cpp17Erasable* from `X` (see 22.2.2.5, below).

`typename X::reference`

4 *Result*: `T&`

`typename X::const_reference`

5 *Result*: `const T&`

`typename X::iterator`

6 *Result*: 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`.

`typename X::const_iterator`

7 *Result*: A type that meets the requirements of a constant iterator and those of a forward iterator with value type `T`.

`typename X::difference_type`

8 *Result*: A signed integer type, identical to the difference type of `X::iterator` and `X::const_iterator`.

`typename X::size_type`

9 *Result*: An unsigned integer type that can represent any non-negative value of `X::difference_type`.

```
X u;
X u = X();
```

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

11 *Complexity*: Constant.

```
X u(a);
X u = a;
```

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

13 *Postconditions*: `u == a`

14 *Complexity*: Linear.

```
X u(rv);
X u = rv;
```

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

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

`a = rv`

17 *Result*: `X&`.

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

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

20 *Complexity*: Linear.

`a.~X()`

21 *Result*: `void`

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

23 *Complexity*: Linear.

`a.begin()`

24 *Result*: `iterator`; `const_iterator` for constant `a`.

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

26 *Complexity*: Constant.

a.end()

27 *Result*: iterator; const_iterator for constant a.

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

29 *Complexity*: Constant.

a.cbegin()

30 *Result*: const_iterator.

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

32 *Complexity*: Constant.

a.cend()

33 *Result*: const_iterator.

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

35 *Complexity*: Constant.

i <=> j

36 *Result*: strong_ordering.

37 *Constraints*: X::iterator meets the random access iterator requirements.

38 *Complexity*: Constant.

a == b

39 *Preconditions*: T meets the *Cpp17EqualityComparable* requirements.

40 *Result*: Convertible to bool.

41 *Value*: equal(a.begin(), a.end(), b.begin(), b.end())

[*Note 1*: The algorithm equal is defined in 25.6.11. — *end note*]

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

43 *Remarks*: == is an equivalence relation.

a != b

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

a.swap(b)

45 *Result*: void

46 *Effects*: Exchanges the contents of a and b.

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

swap(a, b)

48 *Effects*: Equivalent to a.swap(b).

r = a

49 *Result*: X&.

50 *Postconditions*: r == a.

51 *Complexity*: Linear.

a.size()

52 *Result*: size_type.

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

54 *Complexity*: Constant.

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

a.max_size()

56 *Result*: size_type.

57 *Returns*: distance(begin(), end()) for the largest possible container.

Complexity: Constant.

a.empty()

58 *Result*: Convertible to bool.

59 *Value*: a.begin() == a.end()

60 *Complexity*: Constant.

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

62 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.

63 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

(63.1) — allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value,

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

(63.3) — 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.

64 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.

`typename X::reverse_iterator`

2 *Result*: The type `reverse_iterator<X::iterator>`, an iterator type whose value type is `T`.

`typename X::const_reverse_iterator`

3 *Result*: The type `reverse_iterator<X::const_iterator>`, a constant iterator type whose value type is `T`.

`a.rbegin()`

4 *Result*: `reverse_iterator`; `const_reverse_iterator` for constant `a`.

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

6 *Complexity*: Constant.

`a.rend()`

7 *Result*: `reverse_iterator`; `const_reverse_iterator` for constant `a`.

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

9 *Complexity*: Constant.

`a.crbegin()`

10 *Result*: `const_reverse_iterator`.

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

12 *Complexity*: Constant.

`a.crend()`

13 *Result*: `const_reverse_iterator`.

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

15 *Complexity*: Constant.

16 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:

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

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

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

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

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

(16.6) — 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*]

17 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.

18 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
```

2 *Result*: *`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 or is a specialization of `basic_string`, 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, statements, and expressions are well-formed and have the specified semantics.

```
typename X::allocator_type
```

4 *Result*: `A`

5 *Preconditions*: `allocator_type::value_type` is the same as `X::value_type`.

```
c.get_allocator()
```

6 *Result*: `A`

7 *Complexity*: Constant.

```
X u;
X u = X();
```

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

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

10 *Complexity*: Constant.

```
X u(m);
```

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

12 *Complexity*: Constant.

```
X u(t, m);
```

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

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

15 *Complexity*: Linear.

```
X u(rv);
```

16 *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.

17 *Complexity*: Constant.

```
X u(rv, m);
```

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

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

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

```
a = t
```

21 *Result*: `X&`.

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

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

24 *Complexity*: Linear.

```
a = rv
```

25 *Result*: `X&`.

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

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

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

29 *Complexity*: Linear.

`a.swap(b)`

30 *Result*: `void`

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

32 *Complexity*: Constant.

33 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:

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

(33.2) — 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);`

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);`

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)`

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

`a = il`

13 *Result*: `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)`

17 *Result*: `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)`

21 *Result*: `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)`

25 *Result*: `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)`

29 *Result*: `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)`

33 *Result*: `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)`

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

`a.erase(q)`

38 *Result*: `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)`

42 *Result*: `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()`

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)`

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)`

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

`a.assign(n, t)`

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()`

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()`

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

63 *Effects*: Equivalent to:

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

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

`a.emplace_front(args)`

65 *Result*: `reference`

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

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

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

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

a.emplace_back(args)

70 *Result*: reference

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

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

73 *Returns*: a.back().

74 *Remarks*: Required for deque, list, and vector.

a.push_front(t)

75 *Result*: void

76 *Preconditions*: T is *Cpp17CopyInsertable* into X.

77 *Effects*: Prepends a copy of t.

78 *Remarks*: Required for deque, forward_list, and list.

a.push_front(rv)

79 *Result*: void

80 *Preconditions*: T is *Cpp17MoveInsertable* into X.

81 *Effects*: Prepends a copy of rv.

82 *Remarks*: Required for deque, forward_list, and list.

a.push_back(t)

83 *Result*: void

84 *Preconditions*: T is *Cpp17CopyInsertable* into X.

85 *Effects*: Appends a copy of t.

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

a.push_back(rv)

87 *Result*: void

88 *Preconditions*: T is *Cpp17MoveInsertable* into X.

89 *Effects*: Appends a copy of rv.

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

a.pop_front()

91 *Result*: void

92 *Preconditions*: a.empty() is false.

93 *Effects*: Destroys the first element.

94 *Remarks*: Required for deque, forward_list, and list.

a.pop_back()

95 *Result*: void

96 *Preconditions*: a.empty() is false.

97 *Effects*: Destroys the last element.

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

a[n]

99 *Result*: reference; const_reference for constant a

100 *Returns*: *(a.begin() + n)

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

`a.at(n)`

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

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

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

105 *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 77.

Table 77: Container types with compatible nodes [tab:container.node.compat]

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

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

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

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

```
template<unspecified>
class node-handle {
public:
  // These type declarations are described in 22.2.7 and 22.2.8.
  using value_type     = see below;      // not present for map containers
  using key_type       = see below;      // not present for set containers
  using mapped_type    = see below;      // not present for set containers
  using allocator_type = see below;

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

  typename ator_traits::template
    rebind_traits<container_node_type>::pointer ptr_;         // exposition only
  optional<allocator_type> alloc_;                            // exposition only

public:
  // 22.2.5.2, constructors, copy, and assignment
  constexpr node-handle() noexcept : ptr_(), alloc_() {}
  node-handle(node-handle&&) noexcept;
  node-handle& operator=(node-handle&&);

  // 22.2.5.3, destructor
  ~node-handle();
```

6 *Throws*: Nothing.

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

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

8 *Preconditions*: `empty() == false`.

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

10 *Throws*: Nothing.

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

11 *Preconditions*: `empty() == false`.

12 *Returns*: `*alloc_`.

13 *Throws*: Nothing.

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

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

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

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

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 77),

(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_transparent` 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(x, kl)`, with `x` 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, x)`, with `x` the key value of `e` and `e` in `a`;

(7.21) — `ke` is a value such that `a` is partitioned with respect to `c(x, ke)` and `!c(ke, x)`, with `c(x, ke)` implying `!c(ke, x)` and with `x` the key value of `e` and `e` in `a`;

(7.22) — `kx` is a value such that

(7.22.1) — `a` is partitioned with respect to `c(x, kx)` and `!c(kx, x)`, with `c(x, kx)` implying `!c(kx, x)` and with `x` the key value of `e` and `e` in `a`, and

(7.22.2) — `kx` is not convertible to either `iterator` or `const_iterator`; and

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

(7.24) — `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 types, 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*]

```
typename X::key_type
```

9 *Result*: `Key`.

```
typename X::mapped_type
```

10 *Result*: `T`.

11 *Remarks*: For `map` and `multimap` only.

```
typename X::value_type
```

12 *Result*: `Key` for `set` and `multiset` only; `pair<const Key, T>` for `map` and `multimap` only.

13 *Preconditions*: `X::value_type` is *Cpp17Erasable* from `X`.

```
typename X::key_compare
```

14 *Result*: `Compare`.

15 *Preconditions*: `key_compare` is *Cpp17CopyConstructible*.

```
typename X::value_compare
```

16 *Result*: 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`.

```
typename X::node_type
```

17 *Result*: 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)
```

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

19 *Complexity*: Constant.

```
X u = X();
X u;
```

20 *Preconditions*: `key_compare` meets the *Cpp17DefaultConstructible* requirements.

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

22 *Complexity*: Constant.

```
X(i, j, c)
```

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

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

25 *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)
```

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

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

28 *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)

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

X(il)

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

a = il

31 *Result*: X&

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

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

34 *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()

35 *Result*: X::key_compare

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

37 *Complexity*: Constant.

b.value_comp()

38 *Result*: X::value_compare

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

40 *Complexity*: Constant.

a_uniq.emplace(args)

41 *Result*: pair<iterator, bool>

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

43 *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.

44 *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.

45 *Complexity*: Logarithmic.

a_eq.emplace(args)

46 *Result*: iterator

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

48 *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.

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

50 *Complexity*: Logarithmic.

a.emplace_hint(p, args)

51 *Result*: iterator

52 *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.

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

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

a_uniq.insert(t)

55 *Result*: pair<iterator, bool>

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 in the container with key equivalent to the key of `t`.

58 *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`.

59 *Complexity*: Logarithmic.

`a_eq.insert(t)`

60 *Result*: `iterator`

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

62 *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.

63 *Complexity*: Logarithmic.

`a.insert(p, t)`

64 *Result*: `iterator`

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

66 *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`.

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

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

`a.insert(i, j)`

69 *Result*: `void`

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

71 *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.

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

`a.insert(il)`

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

`a_uniq.insert(nh)`

74 *Result*: `insert_return_type`

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

76 *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()`.

77 *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)`

78 *Result*: `iterator`

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

80 *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.

81 *Postconditions*: `nh` is empty.

82 *Complexity*: Logarithmic.

`a.insert(p, nh)`

83 *Result*: `iterator`

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

85 *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`.

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

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

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

`a.extract(k)`

89 *Result*: `node_type`

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

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

92 *Complexity*: log(`a.size()`)

`a_tran.extract(kx)`

93 *Result*: `node_type`

94 *Effects*: Removes the first element in the container with key `r` such that `!c(r, kx) && !c(kx, r)` is `true`.

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

96 *Complexity*: log(`a_tran.size()`)

`a.extract(q)`

97 *Result*: `node_type`

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

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

100 *Complexity*: Amortized constant.

`a.merge(a2)`

101 *Result*: `void`

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

103 *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`.

104 *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`.

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

106 *Complexity*: N log(`a.size()`+N), where N has the value `a2.size()`.

`a.erase(k)`

107 *Result*: `size_type`

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

109 *Returns*: The number of erased elements.

110 *Complexity*: $\log($`a.size()`$) +$ `a.count(k)`

`a_tran.erase(kx)`

111 *Result*: `size_type`

112 *Effects*: Erases all elements in the container with key `r` such that `!c(r, kx) && !c(kx, r)` is `true`.

113 *Returns*: The number of erased elements.

114 *Complexity*: $\log($`a_tran.size()`$) +$ `a_tran.count(kx)`

`a.erase(q)`

115 *Result*: `iterator`

116 *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()`.

117 *Complexity*: Amortized constant.

`a.erase(r)`

118 *Result*: `iterator`

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

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

121 *Complexity*: Amortized constant.

`a.erase(q1, q2)`

122 *Result*: `iterator`

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

124 *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.

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

`a.clear()`

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

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

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

`b.find(k)`

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

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

131 *Complexity*: Logarithmic.

`a_tran.find(ke)`

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

133 *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.

134 *Complexity*: Logarithmic.

`b.count(k)`

135 *Result*: `size_type`

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

137 *Complexity*: $\log($`b.size()`$) +$ `b.count(k)`

a_tran.count(ke)

138 *Result*: size_type

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

140 *Complexity*: log(a_tran.size()) + a_tran.count(ke)

b.contains(k)

141 *Result*: bool

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

a_tran.contains(ke)

143 *Result*: bool

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

b.lower_bound(k)

145 *Result*: iterator; const_iterator for constant b.

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

147 *Complexity*: Logarithmic.

a_tran.lower_bound(kl)

148 *Result*: iterator; const_iterator for constant a_tran.

149 *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.

150 *Complexity*: Logarithmic.

b.upper_bound(k)

151 *Result*: iterator; const_iterator for constant b.

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

153 *Complexity*: Logarithmic,

a_tran.upper_bound(ku)

154 *Result*: iterator; const_iterator for constant a_tran.

155 *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.

156 *Complexity*: Logarithmic.

b.equal_range(k)

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

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

159 *Complexity*: Logarithmic.

a_tran.equal_range(ke)

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

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

162 *Complexity*: Logarithmic.

163 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.

164 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.

165 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
```

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

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

167 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.

168 The member function templates `find`, `count`, `contains`, `lower_bound`, `upper_bound`, `equal_range`, `erase`, and `extract` shall not participate in overload resolution unless the *qualified-id* `Compare::is_transparent` is valid and denotes a type (13.10.3). Additionally, the member function templates `extract` and `erase` shall not participate in overload resolution if `is_convertible_v<K&&, iterator> || is_convertible_v<K&&, const_iterator>` is `true`, where `K` is the type substituted as the first template argument.

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

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

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

(169.3) — 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 77),

(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) — kx is a value such that

(10.20.1) — eq(r1, kx) == eq(kx, r1),

(10.20.2) — hf(r1) == hf(kx) if eq(r1, kx) is true,

(10.20.3) — (eq(r1, kx) && eq(r1, r2)) == eq(r2, kx), and

(10.20.4) — kx is not convertible to either iterator or const_iterator,

where r1 and r2 are keys of elements in a_tran,

(10.21) — n denotes a value of type size_type,

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

(10.23) — 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 types, 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*]

typename X::key_type

12 *Result*: Key.

typename X::mapped_type

13 *Result*: T.

14 *Remarks*: For unordered_map and unordered_multimap only.

typename X::value_type

15 *Result*: Key for unordered_set and unordered_multiset only; pair<const Key, T> for unordered_map and unordered_multimap only.

16 *Preconditions*: value_type is *Cpp17Erasable* from X.

typename X::hasher

17 *Result*: Hash.

18 *Preconditions*: Hash is a unary function object type such that the expression hf(k) has type size_t.

typename X::key_equal

19 *Result*: Pred.

20 *Preconditions*: Pred meets the *Cpp17CopyConstructible* requirements. Pred is a binary predicate that takes two arguments of type Key. Pred is an equivalence relation.

typename X::local_iterator

21 *Result*: 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*]

typename X::const_local_iterator

22 *Result*: An iterator type whose category, value type, difference type, and pointer and reference types are the same as X::const_iterator's.

[*Note 5*: A const_local_iterator object can be used to iterate through a single bucket, but cannot be used to iterate across buckets. — *end note*]

typename X::node_type

23 *Result*: 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)`

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

25 *Complexity*: $\mathcal{O}$(`n`)

`X(n, hf)`

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

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

28 *Complexity*: $\mathcal{O}$(`n`)

`X(n)`

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

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

31 *Complexity*: $\mathcal{O}$(`n`)

```
X a = X();
X a;
```

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

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.

33 *Complexity*: Constant.

`X(i, j, n, hf, eq)`

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

35 *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.

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

`X(i, j, n, hf)`

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

38 *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.

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

`X(i, j, n)`

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

41 *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.

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

`X(i, j)`

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

44 *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.

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

`X(il)`

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

`X(il, n)`

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

`X(il, n, hf)`

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

`X(il, n, hf, eq)`

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

`X(b)`

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

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

`a = b`

52 *Result*: `X&`

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

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

`a = il`

55 *Result*: `X&`

56 *Preconditions*: `value_type` is *Cpp17CopyInsertable* into `X` and *Cpp17CopyAssignable*.

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

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

`b.hash_function()`

59 *Result*: `hasher`

60 *Returns*: `b`'s hash function.

61 *Complexity*: Constant.

`b.key_eq()`

62 *Result*: `key_equal`

63 *Returns*: `b`'s key equality predicate.

64 *Complexity*: Constant.

`a_uniq.emplace(args)`

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

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

67 *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`.

68 *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`.

69 *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}($`a_uniq.size()`$)$.

`a_eq.emplace(args)`

70 *Result*: `iterator`

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

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

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

74 *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}($`a_eq.size()`$)$.

`a.emplace_hint(p, args)`

75 *Result*: `iterator`

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

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

78 *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.

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

`a_uniq.insert(t)`

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

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

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

83 *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`.

84 *Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}($`a_uniq.size()`$)$.

`a_eq.insert(t)`

85 *Result*: `iterator`

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

87 *Effects*: Inserts `t`.

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

89 *Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}($`a_eq.size()`$)$.

`a.insert(p, t)`

90 *Result*: `iterator`

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

92 *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.

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

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

`a.insert(i, j)`

95 *Result*: `void`

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

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

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

`a.insert(il)`

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

`a_uniq.insert(nh)`

100 *Result*: `insert_return_type`

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

102 *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()`.

103 *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()`.

104 *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_uniq.size()`).

`a_eq.insert(nh)`

105 *Result*: `iterator`

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

107 *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.

108 *Postconditions*: `nh` is empty.

109 *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_eq.size()`).

`a.insert(q, nh)`

110 *Result*: `iterator`

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

112 *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.

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

114 *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).

`a.extract(k)`

115 *Result*: `node_type`

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

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

118 *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).

`a_tran.extract(kx)`

119 *Result*: `node_type`

120 *Effects*: Removes an element in the container with key equivalent to `kx`.

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

122 *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_tran.size()`).

`a.extract(q)`

123 *Result*: `node_type`

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

Returns: A `node_type` owning that element.

125 *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).

`a.merge(a2)`

126 *Result*: `void`

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

128 *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`.

129 *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.

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

`a.erase(k)`

131 *Result*: `size_type`

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

133 *Returns*: The number of elements erased.

134 *Complexity*: Average case $\mathcal{O}($`a.count(k)`$)$, worst case $\mathcal{O}($`a.size()`$)$.

`a_tran.erase(kx)`

135 *Result*: `size_type`

136 *Effects*: Erases all elements with key equivalent to `kx`.

137 *Returns*: The number of elements erased.

138 *Complexity*: Average case $\mathcal{O}($`a_tran.count(kx)`$)$, worst case $\mathcal{O}($`a_tran.size()`$)$.

`a.erase(q)`

139 *Result*: `iterator`

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

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

142 *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}($`a.size()`$)$.

`a.erase(r)`

143 *Result*: `iterator`

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

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

146 *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}($`a.size()`$)$.

`a.erase(q1, q2)`

147 *Result*: `iterator`

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

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

150 *Complexity*: Average case linear in `distance(q1, q2)`, worst case $\mathcal{O}($`a.size()`$)$.

`a.clear()`

151 *Result*: `void`

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

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

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

`b.find(k)`

155 *Result*: `iterator`; `const_iterator` for const `b`.

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

157 *Complexity*: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}($`b.size()`$)$.

`a_tran.find(ke)`

158 *Result*: `iterator`; `const_iterator` for const `a_tran`.

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

160 *Complexity*: Average case $\mathscr{O}(1)$, worst case $\mathscr{O}($`a_tran.size()`$)$.

`b.count(k)`

161 *Result*: `size_type`

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

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

`a_tran.count(ke)`

164 *Result*: `size_type`

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

166 *Complexity*: Average case $\mathscr{O}($`a_tran.count(ke)`$)$, worst case $\mathscr{O}($`a_tran.size()`$)$.

`b.contains(k)`

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

`a_tran.contains(ke)`

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

`b.equal_range(k)`

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

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

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

`a_tran.equal_range(ke)`

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

173 *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.

174 *Complexity*: Average case $\mathscr{O}($`a_tran.count(ke)`$)$, worst case $\mathscr{O}($`a_tran.size()`$)$.

`b.bucket_count()`

175 *Result*: `size_type`

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

177 *Complexity*: Constant.

`b.max_bucket_count()`

178 *Result*: `size_type`

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

180 *Complexity*: Constant.

`b.bucket(k)`

181 *Result*: `size_type`

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

183 *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())`.

184 *Complexity*: Constant.

`b.bucket_size(n)`

185 *Result*: `size_type`

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

187 *Returns*: The number of elements in the `n`$^{\text{th}}$ bucket.

188 *Complexity*: $\mathscr{O}($`b.bucket_size(n)`$)$

`b.begin(n)`

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

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

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

192 *Complexity*: Constant.

`b.end(n)`

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

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

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

196 *Complexity*: Constant.

`b.cbegin(n)`

197 *Result*: `const_local_iterator`

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

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

200 *Complexity*: Constant.

`b.cend(n)`

201 *Result*: `const_local_iterator`

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

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

204 *Complexity*: Constant.

`b.load_factor()`

205 *Result*: `float`

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

207 *Complexity*: Constant.

`b.max_load_factor()`

208 *Result*: `float`

209 *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.

210 *Complexity*: Constant.

`a.max_load_factor(z)`

211 *Result*: `void`

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

213 *Complexity*: Constant.

`a.rehash(n)`

214 *Result*: `void`

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

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

`a.reserve(n)`

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

218 Two unordered containers `a` and `b` compare equal if `a.size() == b.size()` and, for every equivalent-key group `[Ea1, Ea2)` obtained from `a.equal_range(Ea1)`, there exists an equivalent-key group `[Eb1, Eb2)` obtained from `b.equal_range(Ea1)`, such that `is_permutation(Ea1, Ea2, Eb1, Eb2)` returns `true`. For `unordered_set` and `unordered_map`, the complexity of `operator==` (i.e., the number of calls to the `==` operator of the `value_type`, to the predicate returned by `key_eq()`, and to the hasher returned by `hash_function()`) is proportional to N in the average case and to N^2 in the worst case, where N is `a.size()`. For `unordered_multiset` and `unordered_multimap`, the complexity of `operator==` is proportional to $\sum E_i^2$ in the average case and to N^2 in the worst case, where N is `a.size()`, and E_i is the size of the i^{th} equivalent-key group in `a`. However, if the respective elements of each corresponding pair of equivalent-key groups Ea_i and Eb_i are arranged in the same order (as is commonly the case, e.g., if `a` and `b` are unmodified copies of the same container), then the average-case complexity for `unordered_multiset` and `unordered_multimap` becomes proportional to N (but worst-case complexity remains $\mathcal{O}(N^2)$, e.g., for a pathologically bad hash function). The behavior of a program that uses `operator==` or `operator!=` on unordered containers is undefined unless the `Pred` function object has the same behavior for both containers and the equality comparison function for `Key` is a refinement[215] of the partition into equivalent-key groups produced by `Pred`.

219 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.

220 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.

221 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.

222 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.

223 The member function templates `find`, `count`, `equal_range`, `contains`, `extract`, and `erase` 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). Additionally, the member function templates `extract` and `erase` shall not participate in overload resolution if `is_convertible_v<K&&, iterator> || is_convertible_v<K&&, const_iterator>` is `true`, where `K` is the type substituted as the first template argument.

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

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

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

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

(224.4) — 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).

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

which shall have the same semantics as the function signatures `intmax_t imaxabs(intmax_t)` and `imaxdiv_t imaxdiv(intmax_t, intmax_t)`, respectively.

SEE ALSO: ISO C 7.8

2 Each of the `PRI` macros listed in this subclause is defined if and only if the implementation defines the corresponding *typedef-name* in 17.4.2. Each of the `SCN` macros listed in this subclause is defined if and only if the implementation defines the corresponding *typedef-name* in 17.4.2 and has a suitable `fscanf` length modifier for the type.

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 130.

Table 130: 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.

typename X::char_type

6 *Result*: charT, the character container type used in the implementation of class template basic_regex.

typename X::string_type

7 *Result*: basic_string<charT>

typename X::locale_type

8 *Result*: A copy constructible type that represents the locale used by the traits class.

typename X::char_class_type

9 *Result*: A bitmask type (16.3.3.3.4) representing a particular character classification.

X::length(p)

10 *Result*: size_t

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

12 *Complexity*: Linear in i.

v.translate(c)

13 *Result*: X::char_type

14 *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)

15 *Result*: X::char_type

16 *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)

17 *Result*: X::string_type

18 *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)

19 *Result*: X::string_type

20 *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)

21 *Result*: X::string_type

22 *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)

23 *Result*: X::char_class_type

24 *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