P2416R1: 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.

Changes vs. R0

- LWG feedback: Presented member typedef requirements with `typename`, but otherwise the same as member function requirements.
- Add “Result” to `[structure.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 and value category of the expression.

(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.

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>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>22.3</td>
<td>Sequence containers &lt;array&gt;, &lt;deque&gt;, &lt;forward_list&gt;, &lt;list&gt;, &lt;vector&gt;</td>
</tr>
<tr>
<td>22.4</td>
<td>Associative containers &lt;map&gt;, &lt;set&gt;</td>
</tr>
<tr>
<td>22.5</td>
<td>Unordered associative containers &lt;unordered_map&gt;, &lt;unordered_set&gt;</td>
</tr>
<tr>
<td>22.6</td>
<td>Container adaptors &lt;queue&gt;, &lt;stack&gt;</td>
</tr>
<tr>
<td>22.7</td>
<td>Views &lt;span&gt;</td>
</tr>
</tbody>
</table>

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 rval 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.

typedef X::value_type

2 Result: T
© ISO/IEC

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

typename X::reference
  Result: T&

typename X::const_reference
  Result: const T&

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

typename X::size_type
  Result: An unsigned integer type that can represent any non-negative value of X::difference_type.

X u;
X u = X();
  Postconditions: u.empty()
  Complexity: Constant.

X u(a);
X u = a;
  Preconditions: T is Cpp17CopyInsertable into X (see below).
  Postconditions: u == a
  Complexity: Linear.

X u(rv);
X u = rv;
  Postconditions: u is equal to the value that rv had before this construction.
  Complexity: Linear for array and constant for all other standard containers.

a = rv
  Result: An lvalue of type X.
  Effects: All existing elements of a are either move assigned to or destroyed.
  Postconditions: If a and rv do not refer to the same object, a is equal to the value that rv had before
  this assignment.
  Complexity: Linear.

a.~X()
  Result: void
  Effects: Destroys every element of a; any memory obtained is deallocated.
  Complexity: Linear.

a.begin()
  Result: A prvalue of type iterator; const_iterator for constant a.
  Value: An iterator referring to the first element in the container.
  Complexity: Constant.
a.end()

Result: A prvalue of type iterator; const_iterator for constant a.
Value: An iterator which is the past-the-end value for the container.
Complexity: Constant.

a.cbegin()

Result: A prvalue of type const_iterator.
Value: const_cast<X const&>(a).begin()
Complexity: Constant.

a.cend()

Result: A prvalue of type const_iterator.
Value: const_cast<X const&>(a).end()
Complexity: Constant.

i <= j

Result: A prvalue of type strong_ordering.
Constraints: X::iterator meets the random access iterator requirements.
Complexity: Constant.

a == b

Preconditions: T meets the Cpp17EqualityComparable requirements.
Result: Convertible to bool.
Value: equal(a.begin(), a.end(), b.begin(), b.end())

[Note 1: The algorithm equal is defined in 25.6.11. — end note]
Complexity: Constant if a.size() != b.size(), linear otherwise.
Remarks: == is an equivalence relation.

a != b

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

a.swap(b)

Result: void
Effects: Exchanges the contents of a and b.
Complexity: Linear for array and constant for all other standard containers.

swap(a, b)

Effects: Equivalent to a.swap(b).

r = a

Result: An lvalue of type X.
Postconditions: r == a.
Complexity: Linear.

a.size()

Result: A prvalue of type size_type.
Value: distance(a.begin(), a.end()), i.e. the number of elements in the container.
Complexity: Constant.
Remarks: The number of elements is defined by the rules of constructors, inserts, and erases.
a.max_size()

56 Result: A prvalue of type size_type.
57 Returns: distance(begin(), end()) for the largest possible container.
58 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.

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

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.

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

typename X::const_reverse_iterator
Result: The type reverse_iterator<X::const_iterator>, a constant iterator type whose value type is T.

a.rbegin()
Result: A prvalue of type reverse_iterator; const_reverse_iterator for constant a.
Value: reverse_iterator(end())
Complexity: Constant.

a.rend()
Result: A prvalue of type reverse_iterator; const_reverse_iterator for constant a.
Value: reverse_iterator(begin())
Complexity: Constant.

a.crbegin()
Result: A prvalue of type const_reverse_iterator.
Value: const_cast<X const&>(a).rbegin()
Complexity: Constant.

a.crend()
Result: A prvalue of type const_reverse_iterator.
Value: const_cast<X const&>(a).rend()
Complexity: Constant.
```

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]

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.

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

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.

\[ a \leftrightarrow b \]

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

** Preconditions:** Either `\leftrightarrow` 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.

**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]

**Complexity:** Linear.

### 22.2.2.5 Allocator-aware containers

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.

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:

1. `T` is **Cpp17DefaultInsertable into X** means that the following expression is well-formed:
   
   \[ \text{allocator_traits}<A>::construct(m, p) \]

2. An element of `X` is **default-inserted** if it is initialized by evaluation of the expression
   
   \[ \text{allocator_traits}<A>::construct(m, p) \]

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

3. `T` is **Cpp17MoveInsertable into X** means that the following expression is well-formed:
   
   \[ \text{allocator_traits}<A>::construct(m, p, rv) \]

   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]

4. `T` is **Cpp17CopyInsertable into X** means that, in addition to 1 being Cpp17MoveInsertable into X, the following expression is well-formed:
   
   \[ \text{allocator_traits}<A>::construct(m, p, v) \]

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

5. `T` is **Cpp17EmplaceConstructible into X from args**, for zero or more arguments `args`, means that the following expression is well-formed:
   
   \[ \text{allocator_traits}<A>::construct(m, p, args) \]

6. `T` is **Cpp17Erasable from X** means that the following expression is well-formed:
   
   \[ \text{allocator_traits}<A>::destroy(m, p) \]

   [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]

In this subclause,

1. `X` denotes an allocator-aware container class with a `value_type` of `T` using an allocator of type `A`,
2. `u` denotes a variable,
3. `a` and `b` denote non-const lvalues of type `X`,
4. `c` denotes an lvalue of type `const X`,

§ 22.2.2.5
— 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
  Result: A
  Preconditions: allocator_type::value_type is the same as X::value_type.

c.get_allocator()
  Result: A
  Complexity: Constant.
X u;
X u = X();
  Preconditions: A meets the Cpp17DefaultConstructible requirements.
  Postconditions: u.empty() returns true, u.get_allocator() == A().
  Complexity: Constant.
X u(m);
  Postconditions: u.empty() returns true, u.get_allocator() == m.
  Complexity: Constant.
X u(t, m);
  Preconditions: T is Cpp17CopyInsertable into X.
  Postconditions: u == t, u.get_allocator() == m
  Complexity: Linear.
X u(rv);
  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.
  Complexity: Constant.
X u(rv, m);
  Preconditions: T is Cpp17MoveInsertable into X.
  Postconditions: u has the same elements, or copies of the elements, that rv had before this construction, u.get_allocator() == m.
  Complexity: Constant if m == rv.get_allocator(), otherwise linear.
```

```
a = t
  Result: An lvalue of type X.
  Preconditions: T is Cpp17CopyInsertable into X and Cpp17CopyAssignable.
  Postconditions: a == t is true.
  Complexity: Linear.
```

```
a = rv
  Result: An lvalue of type X.
  Preconditions: If allocator_traits<allocator_type>::propagate_on_container_move_assignment::value is false, T is Cpp17MoveInsertable into X and Cpp17MoveAssignble.
  Effects: All existing elements of a are either move assigned to or destroyed.
```
Postconditions: If a and rv do not refer to the same object, a is equal to the value that rv had before this assignment.

Complexity: Linear.

a.swap(b)

Result: void

Effects: Exchanges the contents of a and b.

Complexity: Constant.

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

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.

[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

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

[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]

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,
— q denotes a valid dereferenceable constant iterator to a,

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

— t denotes an lvalue or a const rvalue of X::value_type, and

— rv denotes a non-const rvalue of X::value_type.

— Args denotes a template parameter pack;

— args denotes a function parameter pack with the pattern Args&&.

The complexities of the expressions are sequence dependent.

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);`

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

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

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

`X u(i, j);`

**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.

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

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

`X(il)`

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

`a = il`

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

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

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

**Returns:** *this.

`a.emplace(p, args)`

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

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

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

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

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

`a.insert(p, t)`

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

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

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

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

`a.insert(p, rv)`

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

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

**Effects:** Inserts a copy of rv before p.
Returns: An iterator that points to the copy ofrv inserted intoa.

\texttt{\textbf{a.insert}(p, n, t)}

Result: A prvalue of type iterator.

Preconditions: T is Cpp17CopyInsertable into X and Cpp17CopyAssignable.

Effects: Inserts n copies of t before p.

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

\texttt{\textbf{a.insert}(p, i, j)}

Effects: Equivalent to \texttt{a.insert(p, il.begin(), il.end())}.

\texttt{\textbf{a.erase}(q)}

Preconditions: For vector and deque, T is Cpp17MoveAssignabler.

Effects: Erases the element pointed to by q.

Returns: An iterator that points to the element immediately following q prior to the element being erased. If no such element exists, \texttt{a.end()} is returned.

\texttt{\textbf{a.erase}(q1, q2)}

Preconditions: For vector and deque, T is Cpp17MoveAssignabler.

Effects: Erases the elements in the range \([q1, q2)\).

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

\texttt{\textbf{a.clear}()}\n
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.

Preconditions: \texttt{a.empty()} is true.

Complexity: Linear.

\texttt{\textbf{a.assign}(i, j)}

Effects: Equivalent to \texttt{a.assign(il.begin(), il.end())}.
\textbf{a.assign(n, t)}

\textit{Result:} void

\textit{Preconditions:} \textit{T} is \textit{Cpp17CopyInsertable} into \textit{X} and \textit{Cpp17CopyAssignable}. \textit{t} is not a reference into \textit{a}.

\textit{Effects:} Replaces elements in \textit{a} with \textit{n} copies of \textit{t}. Invalidates all references, pointers and iterators referring to the elements of \textit{a}. For \textit{vector} and \textit{deque}, also invalidates the past-the-end iterator.

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

1. If the constructor
   \begin{verbatim}
   template<class InputIterator>
   X(InputIterator first, InputIterator last,
     const allocator_type& alloc = allocator_type());
   \end{verbatim}
   is called with a type \textit{InputIterator} that does not qualify as an input iterator, then the constructor shall not participate in overload resolution.

2. If the member functions of the forms:
   \begin{verbatim}
   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
   \end{verbatim}
   are called with a type \textit{InputIterator} that does not qualify as an input iterator, then these functions shall not participate in overload resolution.

3. A deduction guide for a sequence container shall not participate in overload resolution if it has an \textit{InputIterator} template parameter and a type that does not qualify as an input iterator is deduced for that parameter, or if it has an \textit{Allocator} template parameter and a type that does not qualify as an allocator is deduced for that parameter.

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.

\textbf{a.front()}

\textit{Result:} reference; const reference for constant \textit{a}.

\textit{Returns:} *a.begin()

\textit{Remarks:} Required for \textit{basic_string}, \textit{array}, \textit{deque}, \textit{forward_list}, \textit{list}, and \textit{vector}.

\textbf{a.back()}

\textit{Effects:} Equivalent to:
   \begin{verbatim}
   auto tmp = a.end();
   --tmp;
   return *tmp;
   \end{verbatim}

\textit{Remarks:} Required for \textit{basic_string}, \textit{array}, \textit{deque}, \textit{list}, and \textit{vector}.

\textbf{a.emplace_front(args)}

\textit{Result:} reference

\textit{Preconditions:} \textit{T} is \textit{Cpp17EmplaceConstructible} into \textit{X} from \textit{args}.

\textit{Effects:} Prepends an object of type \textit{T} constructed with \textbf{std::forward<Args>(args)....}

\textit{Returns:} a.front().

\textit{Remarks:} Required for \textit{deque}, \textit{forward_list}, and \textit{list}.

\textbf{a.emplace_back(args)}

\textit{Result:} reference
Preconditions: T is Cpp17EmplaceConstructible into X from args. For vector, T is also Cpp17MoveInsertable into X.

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

Returns: a.back().

Remarks: Required for deque, list, and vector.

a.push_front(t)

Result: void

Preconditions: T is Cpp17CopyInsertable into X.

Effects: Prepends a copy of t.

Remarks: Required for deque, forward_list, and list.

a.push_front(rv)

Result: void

Preconditions: T is Cpp17MoveInsertable into X.

Effects: Prepends a copy of rv.

Remarks: Required for deque, forward_list, and list.

a.push_back(t)

Result: void

Preconditions: T is Cpp17CopyInsertable into X.

Effects: Appends a copy of t.

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

a.push_back(rv)

Result: void

Preconditions: T is Cpp17MoveInsertable into X.

Effects: Appends a copy of rv.

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

a.pop_front()

Result: void

Preconditions: a.empty() is false.

Effects: Destroys the first element.

Remarks: Required for deque, forward_list, and list.

a.pop_back()

Result: void

Preconditions: a.empty() is false.

Effects: Destroys the last element.

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

a[n]

Result: reference; const_reference for constant a

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

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

a.at(n)

Result: reference; const_reference for constant a

Returns: *(a.begin() + n)
Throws: out_of_range if n >= a.size().
Remarks: Required for basic_string, array, deque, and vector.

22.2.5 Node handles

22.2.5.1 Overview

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

<table>
<thead>
<tr>
<th>Container Type</th>
<th>Compatible Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>map&lt;K, T, C1, A&gt;</td>
<td>map&lt;K, T, C2, A&gt;</td>
</tr>
<tr>
<td>map&lt;K, T, C1, A&gt;</td>
<td>multimap&lt;K, T, C2, A&gt;</td>
</tr>
<tr>
<td>set&lt;K, C1, A&gt;</td>
<td>set&lt;K, C2, A&gt;</td>
</tr>
<tr>
<td>set&lt;K, C1, A&gt;</td>
<td>multiset&lt;K, C2, A&gt;</td>
</tr>
<tr>
<td>unordered_map&lt;K, T, H1, E1, A&gt;</td>
<td>unordered_map&lt;K, T, H2, E2, A&gt;</td>
</tr>
<tr>
<td>unordered_map&lt;K, T, H1, E1, A&gt;</td>
<td>unordered_multimap&lt;K, T, H2, E2, A&gt;</td>
</tr>
<tr>
<td>unordered_set&lt;K, H1, E1, A&gt;</td>
<td>unordered_set&lt;K, H2, E2, A&gt;</td>
</tr>
<tr>
<td>unordered_set&lt;K, H1, E1, A&gt;</td>
<td>unordered_multiset&lt;K, H2, E2, A&gt;</td>
</tr>
</tbody>
</table>

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.

Class node_handle is for exposition only.

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::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();
  // 22.2.5.4, observers
  value_type& value() const; // not present for map containers
  key_type& key() const; // not present for set containers
  mapped_type& mapped() const; // not present for set containers

§ 22.2.5.1 828
mapped_type& mapped() const;
  
  **Preconditions:** empty() == false.
  
  **Returns:** A reference to the mapped_type member of the value_type subobject in the container_-
  node_type object pointed to by ptr_.
  
  **Throws:** Nothing.

allocator_type get_allocator() const;
  
  **Preconditions:** empty() == false.
  
  **Returns:** *alloc_.
  
  **Throws:** Nothing.

explicit operator bool() const noexcept;
  
  **Returns:** ptr_ != nullptr.

[[nodiscard]] bool empty() const noexcept;
  
  **Returns:** ptr_ == nullptr.

### 22.2.5.5 Modifiers

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

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

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

### 22.2.6 Insert return type

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.

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

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

#### 22.2.7.1 General

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

Each associative container is parameterized on `Key` and an ordering relation `Compare` that induces a strict
weak ordering (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.

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(r, kl)`, with `r` the key value of `e` and `e` in `a`;
(7.20) — `ku` is a value such that `a` is partitioned with respect to `!c(ku, r)`;
(7.21) — `ke` is a value such that `a` is partitioned with respect to `c(r, ke)` and `!c(ke, r)`, with `c(r, ke)` implying `!c(ke, r)`;
(7.22) — `kx` is a value such that
(7.22.1) — `a` is partitioned with respect to `c(r, kx)` and `!c(kx, r)`, with `c(r, kx)` implying `!c(kx, r)`, 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]

```cpp
typename X::key_type
   Result: Key.

typename X::mapped_type
   Result: T.
   Remarks: For map and multimap only.

typename X::value_type
   Result: Key for set and multiset only; pair<const Key, T> for map and multimap only.
   Preconditions: X::value_type is Cpp17Erasable from X.

typename X::key_compare
   Result: Compare.
   Preconditions: key_compare is Cpp17CopyConstructible.

typename X::value_compare
   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
   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.
```

\texttt{X(c)}

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

X u = X();
X u;

Preconditions: key_compare meets the Cpp17DefaultConstructible requirements.

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

Complexity: Constant.

X(i, j, c)

Preconditions: value_type is Cpp17EmplaceConstructible into X from *i.

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

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)

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

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

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)

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

**Effects**: Equivalent to \(X(il.\text{begin()}\), il.\text{end()}\).

\(a = il\)

**Result**: \(X\)

**Preconditions**: \(\text{value\_type is } Cpp17\text{CopyInsertable into } X \text{ and } Cpp17\text{CopyAssignble.} \)

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

**Complexity**: \(N \log N\) in general, where \(N\) has the value \(il.\text{size()} + a.\text{size()\};\) linear if \([il.\text{begin()}\), il.\text{end()}\]) is sorted with \(value\_\text{comp()}\).

\(b.\text{key\_comp()}\)

**Result**: \(X::\text{key\_compare}\)

**Returns**: The comparison object out of which \(b\) was constructed.

**Complexity**: Constant.

\(b.\text{value\_comp()}\)

**Result**: \(X::\text{value\_compare}\)

**Returns**: An object of \(\text{value\_compare}\) constructed out of the comparison object.

**Complexity**: Constant.

\(a\_\text{uniq.emplace(args)}\)

**Result**: \(\text{pair<iterator, bool>}\)

**Preconditions**: \(\text{value\_type is } Cpp17\text{EmplaceConstructible into } X \text{ from args.} \)

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

**Returns**: The \(\text{bool}\) component of the returned pair is \(\text{true}\) 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 \(t\).

**Complexity**: Logarithmic.

\(a\_\text{eq.emplace(args)}\)

**Result**: iterator

**Preconditions**: \(\text{value\_type is } Cpp17\text{EmplaceConstructible into } X \text{ from args.} \)

**Effects**: Inserts a \(\text{value\_type}\) object \(t\) constructed with \(\text{std::forward<Args>(args)}\).... If a range containing elements equivalent to \(t\) exists in \(a\_\text{eq}\), \(t\) is inserted at the end of that range.

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

**Complexity**: Logarithmic.

\(a.\text{emplace\_hint(p, args)}\)

**Result**: iterator

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

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

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

\(a\_\text{uniq.insert(t)}\)

**Result**: \(\text{pair<iterator, bool>}\)

**Preconditions**: If \(t\) is a non-const rvalue, \(\text{value\_type is } Cpp17\text{MoveInsertable into } X\); otherwise, \(\text{value\_type is } Cpp17\text{CopyInsertable into } X\).

**Effects**: Inserts \(t\) if and only if there is no element in the container with key equivalent to the key of \(t\).
Returns: The \texttt{bool} component of the returned pair is \texttt{true} if and only if the insertion takes place, and the \texttt{iterator} component of the pair points to the element with key equivalent to the key of \( t \).

\textit{Complexity: Logarithmic.}

\texttt{a\_eq.insert(t)}

\textit{Result: iterator}

\textit{Preconditions:} If \( t \) is a non-const rvalue, \texttt{value\_type} is \texttt{Cpp17MoveInsertable} into \( X \); otherwise, \texttt{value\_type} is \texttt{Cpp17CopyInsertable} into \( X \).

\textit{Effects:} Inserts \( t \) and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to \( t \) exists in \texttt{a\_eq}, \( t \) is inserted at the end of that range.

\textit{Complexity: Logarithmic.}

\texttt{a.insert(p, t)}

\textit{Result: iterator}

\textit{Preconditions:} If \( t \) is a non-const rvalue, \texttt{value\_type} is \texttt{Cpp17MoveInsertable} into \( X \); otherwise, \texttt{value\_type} is \texttt{Cpp17CopyInsertable} into \( X \).

\textit{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 \( p \).

\textit{Returns:} An iterator pointing to the element with key equivalent to the key of \( t \).

\textit{Complexity: Logarithmic in general, but amortized constant if \( t \) is inserted right before \( p \).}

\texttt{a.insert(i, j)}

\textit{Result: void}

\textit{Preconditions:} \texttt{value\_type} is \texttt{Cpp17EmplaceConstructible} into \( X \) from \( *i \). Neither \( i \) nor \( j \) are iterators into \( a \).

\textit{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.

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

\texttt{a.insert(il)}

\textit{Effects:} Equivalent to \texttt{a.insert(il.begin(), il.end())}.

\texttt{a\_uniq.insert(nh)}

\textit{Result: insert\_return\_type}

\textit{Preconditions:} \( nh \) is empty or \texttt{a\_uniq.get\_allocator()} == \texttt{nh.get\_allocator()} is \texttt{true}.

\textit{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 \( nh.key() \).

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

\textit{Complexity: Logarithmic.}

\texttt{a\_eq.insert(nh)}

\textit{Result: iterator}

\textit{Preconditions:} \( nh \) is empty or \texttt{a\_eq.get\_allocator()} == \texttt{nh.get\_allocator()} is \texttt{true}.

\textit{Effects:} If \( nh \) is empty, has no effect and returns \texttt{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 \texttt{a\_eq}, the element is inserted at the end of that range.

\textit{Postconditions:} \( nh \) is empty.
Complexity: Logarithmic.

\begin{verbatim}
a.insert(p, nh)
\end{verbatim}

Result: iterator

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

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.

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

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

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

\begin{verbatim}
a.extract(k)
\end{verbatim}

Result: node_type

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

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

Complexity: \text{log}(a.size())

\begin{verbatim}
a_tran.extract(kx)
\end{verbatim}

Result: node_type

Effects: Removes the first element in the container with key r such that \text{!c}(r, kx) && \text{!c}(kx, r) is true.

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

Complexity: \text{log}(a_tran.size())

\begin{verbatim}
a.extract(q)
\end{verbatim}

Result: node_type

Effects: Removes the element pointed to by q.

Returns: A node_type owning that element.

Complexity: Amortized constant.

\begin{verbatim}
a.merge(a2)
\end{verbatim}

Result: void

Preconditions: a.get_allocator() == a2.get_allocator().

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.

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.

Throws: Nothing unless the comparison object throws.

Complexity: \text{N log}(a.size()+N), where N has the value a2.size().

\begin{verbatim}
a.erase(k)
\end{verbatim}

Result: size_type

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

Returns: The number of erased elements.

Complexity: \text{log}(a.size()) + a.count(k)
a_tran.erase(kx)

Result: size_type
Effects: Erases all elements in the container with key r such that \( !c(r, kx) \) \&\& \( !c(kx, r) \) is true.
Returns: The number of erased elements.
Complexity: \( \log(a\_tran.size()) + a\_tran.count(kx) \)

a.erase(q)

Result: iterator
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().
Complexity: Amortized constant.

a.erase(r)

Result: iterator
Effects: Erases the element pointed to by r.
Returns: An iterator pointing to the element immediately following r prior to the element being erased. If no such element exists, returns a.end().
Complexity: Amortized constant.

a.erase(q1, q2)

Result: iterator
Effects: Erases all the elements in the range \([q1, q2)\).
Returns: An iterator pointing to the element pointed to by q2 prior to any elements being erased. If no such element exists, a.end() is returned.
Complexity: \( \log(a.size()) + N \), where \( N \) has the value distance(q1, q2).

a.clear()

Effects: Equivalent to a.erase(a.begin(), a.end()).
Postconditions: a.empty() is true.
Complexity: Linear in a.size().

b.find(k)

Result: iterator; const_iterator for constant b.
Returns: An iterator pointing to an element with the key equivalent to k, or b.end() if such an element is not found.
Complexity: Logarithmic.

a_tran.find(ke)

Result: iterator; const_iterator for constant a_tran.
Returns: An iterator pointing to an element with key r such that \( !c(r, ke) \) \&\& \( !c(ke, r) \) is true, or a_tran.end() if such an element is not found.
Complexity: Logarithmic.

b.count(k)

Result: size_type
Returns: The number of elements with key equivalent to k.
Complexity: \( \log(b.size()) + b.count(k) \)

a_tran.count(ke)

Result: size_type
Returns: The number of elements with key \( r \) such that \( \neg c(r, ke) \land \neg c(ke, r) \).

Complexity: \( \log(a\text{\_tran.size()}) + a\text{\_tran.count(ke)} \)

\[
\text{b.contains(k)}
\]

Result: bool

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

\[
\text{a\_tran.contains(ke)}
\]

Result: bool

Effects: Equivalent to: return a\_tran.find(ke) != a\_tran.end();

\[
\text{b.lower_bound(k)}
\]

Result: iterator; const_iterator for constant b.

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

Complexity: Logarithmic.

\[
\text{a\_tran.lower_bound(kl)}
\]

Result: iterator; const_iterator for constant a\_tran.

Returns: An iterator pointing to the first element with key \( r \) such that \( \neg c(r, kl) \), or a\_tran.end() if such an element is not found.

Complexity: Logarithmic.

\[
\text{b.upper_bound(k)}
\]

Result: iterator; const_iterator for constant b.

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

Complexity: Logarithmic.

\[
\text{a\_tran.upper_bound(ku)}
\]

Result: iterator; const_iterator for constant a\_tran.

Returns: An iterator pointing to the first element with key \( r \) such that \( c(ku, r) \), or a\_tran.end() if such an element is not found.

Complexity: Logarithmic.

\[
\text{b.equal_range(k)}
\]

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

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

Complexity: Logarithmic.

\[
\text{a\_tran.equal_range(ke)}
\]

Result: pair<iterator, iterator>; pair<const_iterator, const_iterator> for constant a\_tran.

Effects: Equivalent to: return make_pair(a\_tran.lower_bound(ke), a\_tran.upper_bound(ke));

Complexity: Logarithmic.

The \text{insert} and \text{emplace} members shall not affect the validity of iterators and references to the container, and the \text{erase} members shall invalidate only iterators and references to the erased elements.

The \text{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 \text{node\_type} is undefined behavior. References and pointers to an element obtained while it is owned by a \text{node\_type} are invalidated if the element is successfully inserted.

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

\[
\text{value_comp}(*j, *i) == \text{false}
\]

For associative containers with unique keys the stronger condition holds:

\[
\text{value_comp}(*i, *j) != \text{false}
\]

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

The member function templates `find`, `count`, `contains`, `lower_bound`, `upper_bound`, `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.

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

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

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

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`.

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.

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`.

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.

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 $k_1$ and $k_2$ in the same container, calling $\text{pred}(k_1, k_2)$ shall always return the same value. For any key $k$ in a container, calling $\text{hash}(k)$ shall always return the same value.

An unordered associative container supports unique keys if it may contain at most one element for each key. Otherwise, it supports equivalent keys. $\text{unordered_set}$ and $\text{unordered_map}$ support unique 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.

For $\text{unordered_set}$ and $\text{unordered_multiset}$ the value type is the same as the key type. For $\text{unordered_map}$ and $\text{unordered_multimap}$ it is $\text{pair<const Key, T}}$.

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 $\text{unordered_multiset}$ and $\text{unordered_multimap}$, rehashing preserves the relative ordering of equivalent elements.

In this subclause,

- $X$ denotes an unordered associative container class,
- $a$ denotes a value of type $X$,
- $a_2$ denotes a value of a type with nodes compatible with type $X$ (Table 77),
- $b$ denotes a possibly const value of type $X$,
- $a_{\text{uniq}}$ denotes a value of type $X$ when $X$ supports unique keys,
- $a_{\text{eq}}$ denotes a value of type $X$ when $X$ supports equivalent keys,
- $a_{\text{tran}}$ denotes a possibly const value of type $X$ when the qualified-ids $X::\text{key_equal}::\text{is\_transparent}$ and $X::\text{hasher}::\text{is\_transparent}$ are both valid and denote types (13.10.3),
- $i$ and $j$ denote input iterators that refer to $\text{value\_type}$,
- $[i, j)$ denotes a valid range,
- $p$ and $q_2$ denote valid constant iterators to $a$,
- $q$ and $q_1$ denote valid dereferenceable constant iterators to $a$,
- $r$ denotes a valid dereferenceable iterator to $a$,
- $[q_1, q_2)$ denotes a valid range in $a$,
- $i$ denotes a value of type $\text{initializer\_list<value\_type>}$,
- $t$ denotes a value of type $X::\text{value\_type}$,
- $k$ denotes a value of type $\text{key\_type}$,
- $hf$ denotes a possibly const value of type $\text{hasher}$,
- $eq$ denotes a possibly const value of type $\text{key\_equal}$,
- $ke$ is a value such that
  - $\text{eq}(r_1, ke) == \text{eq}(ke, r_1)$,
  - $hf(r_1) == hf(ke)$ if $\text{eq}(r_1, ke)$ is true, and
  - $(\text{eq}(r_1, ke) && \text{eq}(r_1, r_2)) == \text{eq}(r_2, ke)$,
where \( r_1 \) and \( r_2 \) are keys of elements in \( a_{\text{tran}} \),

\[
(10.20)
\begin{align*}
&-\text{kx is a value such that} \\
&(10.20.1)\quad \text{eq}(r_1, \text{kx}) == \text{eq}(\text{kx}, r_1), \\
&(10.20.2)\quad \text{hf}(r_1) == \text{hf}(\text{kx}) \text{ if } \text{eq}(r_1, \text{kx}) \text{ is true}, \\
&(10.20.3)\quad (\text{eq}(r_1, \text{kx}) \&\& \text{eq}(r_1, r_2)) == \text{eq}(r_2, \text{kx}), \text{ and} \\
&(10.20.4)\quad \text{kx} \text{ is not convertible to either iterator or const_iterator,}
\end{align*}
\]

where \( r_1 \) and \( r_2 \) are keys of elements in \( a_{\text{tran}} \),

\[
(10.21)
\begin{align*}
&-\text{n denotes a value of type size_type,} \\
&(10.22)\quad \text{z denotes a value of type float, and} \\
&(10.23)\quad \text{nh denotes a non-const rvalue of type X::node_type.}
\end{align*}
\]

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]

\[
\text{typename X::key_type} \\
\quad \text{Result: Key.}
\]

\[
\text{typename X::mapped_type} \\
\quad \text{Result: T.} \\
\quad \text{Remarks: For unordered_map and unordered_multimap only.}
\]

\[
\text{typename X::value_type} \\
\quad \text{Result: Key for unordered_set and unordered_multiset only; pair<const Key, T> for unordered_map and unordered_multimap only.} \\
\quad \text{Preconditions: value_type is Cpp17Erasable from X.}
\]

\[
\text{typename X::hasher} \\
\quad \text{Result: Hash.} \\
\quad \text{Preconditions: Hash is a unary function object type such that the expression hf(k) has type size_t.}
\]

\[
\text{typename X::key_equal} \\
\quad \text{Result: Pred.} \\
\quad \text{Preconditions: Pred meets the Cpp17CopyConstructible requirements. Pred is a binary predicate that takes two arguments of type Key. Pred is an equivalence relation.}
\]

\[
\text{typename X::local_iterator} \\
\quad \text{Result: An iterator type whose category, value type, difference type, and pointer and reference types are the same as X::iterator's.} \\
\quad \text{[Note 4: A local_iterator object can be used to iterate through a single bucket, but cannot be used to iterate across buckets. — end note]}
\]

\[
\text{typename X::const_local_iterator} \\
\quad \text{Result: An iterator type whose category, value type, difference type, and pointer and reference types are the same as X::const_iterator's.} \\
\quad \text{[Note 5: A const_local_iterator object can be used to iterate through a single bucket, but cannot be used to iterate across buckets. — end note]}
\]

\[
\text{typename X::node_type} \\
\quad \text{Result: A specialization of a node-handle class template (22.2.5), such that the public nested types are the same types as the corresponding types in X.}
\]

§ 22.2.8.1

840
X(n, hf, eq)

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

Complexity: \( \Theta(n) \)

X(n, hf)

Preconditions: key_equal meets the Cpp17DefaultConstructible requirements.

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

Complexity: \( \Theta(n) \)

X(n)

Preconditions: hasher and key_equal meet the Cpp17DefaultConstructible requirements.

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

Complexity: \( \Theta(n) \)

X a = X();
X a;

Preconditions: hasher and key_equal meet the Cpp17DefaultConstructible requirements.

Effects: Constructs an empty container with an unspecified number of buckets, using hasher() as the hash function and key_equal() as the key equality predicate.

Complexity: Constant.

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

Preconditions: value_type is Cpp17EmplaceConstructible into X from *i.

Effects: Constructs an empty container with at least \( n \) buckets, using hf as the hash function and eq as the key equality predicate, and inserts elements from \([i, j)\) into it.

Complexity: Average case \( \Theta(N) \) (\( N \) is distance(i, j)), worst case \( \Theta(N^2) \).

X(i, j, n, hf)

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

Effects: Constructs an empty container with at least \( n \) buckets, using hf as the hash function and key_equal() as the key equality predicate, and inserts elements from \([i, j)\) into it.

Complexity: Average case \( \Theta(N) \) (\( N \) is distance(i, j)), worst case \( \Theta(N^2) \).

X(i, j, n)

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

Effects: Constructs an empty container with at least \( n \) buckets, using hasher() as the hash function and key_equal() as the key equality predicate, and inserts elements from \([i, j)\) into it.

Complexity: Average case \( \Theta(N) \) (\( N \) is distance(i, j)), worst case \( \Theta(N^2) \).

X(i, j)

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

Effects: Constructs an empty container with an unspecified number of buckets, using hasher() as the hash function and key_equal() as the key equality predicate, and inserts elements from \([i, j)\) into it.

Complexity: Average case \( \Theta(N) \) (\( N \) is distance(i, j)), worst case \( \Theta(N^2) \).

X(il)

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

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

X(il, n, hf)

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

X(il, n, hf, eq)

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

X(b)

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

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

a = b

Result: X&

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

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

a = il

Result: X&

Preconditions: value_type is Cpp17CopyInsertable into X and Cpp17CopyAssignable.

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

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

b.hash_function()

Result: hasher

Returns: b’s hash function.

Complexity: Constant.

b.key_eq()

Result: key_equal

Returns: b’s key equality predicate.

Complexity: Constant.

a_uniq.emplace(args)

Result: pair<iterator, bool>

Preconditions: value_type is Cpp17EmplaceConstructible into X from args.

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

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.

Complexity: Average case $\Theta(1)$, worst case $\Theta(a_uniq.size())$.

a_eq.emplace(args)

Result: iterator

Preconditions: value_type is Cpp17EmplaceConstructible into X from args.

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

Returns: An iterator pointing to the newly inserted element.

Complexity: Average case $\Theta(1)$, worst case $\Theta(a_eq.size())$.
a.emplace_hint(p, args)

    Result: iterator
    Preconditions: value_type is Cpp17EmplaceConstructible into X from args.
    Effects: Equivalent to a.emplace(std::forward<Args>(args)...).
    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.

    Complexity: Average case $\Theta(1)$, worst case $\Theta(a.size())$.

a_uniq.insert(nh)

    Result: insert_return_type
    Preconditions: nh is empty or a_uniq.get_allocator() == nh.get_allocator() is true.
    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().

    $\S$ 22.2.8.1 843
**Postconditions:** If \( nh \) is empty, \( \text{inserted} \) is false, \( \text{position} \) is \( \text{end()} \), and \( \text{node} \) is empty. Otherwise if the insertion took place, \( \text{inserted} \) is true, \( \text{position} \) points to the inserted element, and \( \text{node} \) is empty; if the insertion failed, \( \text{inserted} \) is false, \( \text{node} \) has the previous value of \( nh \), and \( \text{position} \) points to an element with a key equivalent to \( nh\text{.key()} \).

**Complexity:** Average case \( \Theta(1) \), worst case \( \Theta(a\text{.uniq.size()}) \).

\[ \text{a\text{.eq.insert}}(nh) \]

**Result:** iterator

**Preconditions:** \( nh \) is empty or \( a\text{.eq.get_allocator()} == nh\text{.get_allocator()} \) is true.

**Effects:** If \( nh \) is empty, has no effect and returns \( a\text{.eq.end()} \). Otherwise, inserts the element owned by \( nh \) and returns an iterator pointing to the newly inserted element.

**Postconditions:** \( nh \) is empty.

**Complexity:** Average case \( \Theta(1) \), worst case \( \Theta(a\text{.eq.size()}) \).

\[ \text{a.insert}(q, nh) \]

**Result:** iterator

**Preconditions:** \( nh \) is empty or \( a\text{.get_allocator()} == nh\text{.get_allocator()} \) is true.

**Effects:** If \( nh \) is empty, has no effect and returns \( a\text{.end()} \). Otherwise, inserts the element owned by \( nh \) if and only if there is no element with key equivalent to \( nh\text{.key()} \) 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.

**Returns:** An iterator pointing to the element with key equivalent to \( nh\text{.key()} \).

**Complexity:** Average case \( \Theta(1) \), worst case \( \Theta(a\text{.size()}) \).

\[ \text{a.extract}(k) \]

**Result:** node_type

**Effects:** Removes an element in the container with key equivalent to \( k \).

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

**Complexity:** Average case \( \Theta(1) \), worst case \( \Theta(a\text{.size()}) \).

\[ \text{a\text{.tran.extract}}(kx) \]

**Result:** node_type

**Effects:** Removes an element in the container with key equivalent to \( kx \).

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

**Complexity:** Average case \( \Theta(1) \), worst case \( \Theta(a\text{.tran.size()}) \).

\[ \text{a.extract}(q) \]

**Result:** node_type

**Effects:** Removes the element pointed to by \( q \).

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

**Complexity:** Average case \( \Theta(1) \), worst case \( \Theta(a\text{.size()}) \).

\[ \text{a.merga}(a2) \]

**Result:** void

**Preconditions:** \( a\text{.get_allocator()} == a2\text{.get_allocator()} \).

**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 \( a2 \).
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.

Complexity: Average case \(\Theta(N)\), where \(N\) is \(a2.size()\), worst case \(\Theta(N+a.size() + n)\).

\[ \text{a.erase(k)} \]

Result: size_type

Effects: Erases all elements with key equivalent to \(k\).

Returns: The number of elements erased.

Complexity: Average case \(\Theta(a.count(k))\), worst case \(\Theta(a.size())\).

\[ \text{a Tran.erase(kx)} \]

Result: size_type

Effects: Erases all elements with key equivalent to \(kx\).

Returns: The number of elements erased.

Complexity: Average case \(\Theta(a Tran.count(kx))\), worst case \(\Theta(a Tran.size())\).

\[ \text{a.erase(q)} \]

Result: iterator

Effects: Erases the element pointed to by \(q\).

Returns: The iterator immediately following \(q\) prior to the erasure.

Complexity: Average case \(\Theta(1)\), worst case \(\Theta(a.size())\).

\[ \text{a.erase(r)} \]

Result: iterator

Effects: Erases the element pointed to by \(r\).

Returns: The iterator immediately following \(r\) prior to the erasure.

Complexity: Average case \(\Theta(1)\), worst case \(\Theta(a.size())\).

\[ \text{a.erase(q1, q2)} \]

Result: iterator

Effects: Erases all elements in the range \([q1, q2)\).

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

Complexity: Average case linear in \(\text{distance}(q1, q2)\), worst case \(\Theta(a.size())\).

\[ \text{a.clear()} \]

Result: void

Effects: Erases all elements in the container.

Postconditions: \(a.empty()\) is true.

Complexity: Linear in \(a.size()\).

b.find(k)

Result: iterator; const_iterator for const b.

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

Complexity: Average case \(\Theta(1)\), worst case \(\Theta(b.size())\).

\[ \text{a Tran.find(ke)} \]

Result: iterator; const_iterator for const a Tran.

Returns: An iterator pointing to an element with key equivalent to \(ke\), or \(a Tran.end()\) if no such element exists.
Complexity: Average case $O(1)$, worst case $O(a_{\text{tran}.\size})$.

\textbf{b.count(k)}

\textbf{Result: size_type}
\textbf{Returns:} The number of elements with key equivalent to \texttt{k}.

\textbf{Complexity:} Average case $O(\text{b.count(k)})$, worst case $O(\text{b.size()})$.

\textbf{a_tran.count(ke)}

\textbf{Result: size_type}
\textbf{Returns:} The number of elements with key equivalent to \texttt{ke}.

\textbf{Complexity:} Average case $O(\text{a_tran.count(ke)})$, worst case $O(\text{a_tran.size()})$.

\textbf{b.contains(k)}

\textbf{Effects:} Equivalent to \texttt{b.find(k) != b.end()}.

\textbf{a_tran.contains(ke)}

\textbf{Effects:} Equivalent to \texttt{a_tran.find(ke) != a_tran.end()}.

\textbf{b.equal_range(k)}

\textbf{Result: pair\<iterator, iterator\>; pair\<const_iterator, const_iterator\> for const b.}
\textbf{Returns:} A range containing all elements with keys equivalent to \texttt{k}. Returns \texttt{make_pair(b.end(), b.end())} if no such elements exist.

\textbf{Complexity:} Average case $O(\text{b.count(k)})$, worst case $O(\text{b.size()})$.

\textbf{a_tran.equal_range(ke)}

\textbf{Result: pair\<iterator, iterator\>; pair\<const_iterator, const_iterator\> for const a_tran.}
\textbf{Returns:} A range containing all elements with keys equivalent to \texttt{ke}. Returns \texttt{make_pair(a_tran.end(), a_tran.end())} if no such elements exist.

\textbf{Complexity:} Average case $O(\text{a_tran.count(ke)})$, worst case $O(\text{a_tran.size()})$.

\textbf{b.bucket_count()}

\textbf{Result: size_type}
\textbf{Returns:} The number of buckets that \texttt{b} contains.

\textbf{Complexity:} Constant.

\textbf{b.max_bucket_count()}

\textbf{Result: size_type}
\textbf{Returns:} An upper bound on the number of buckets that \texttt{b} can ever contain.

\textbf{Complexity:} Constant.

\textbf{b.bucket(k)}

\textbf{Result: size_type}
\textbf{Preconditions:} \texttt{b.bucket_count() > 0}.
\textbf{Returns:} The index of the bucket in which elements with keys equivalent to \texttt{k} would be found, if any such element existed. The return value is in the range $[0, \text{b.bucket_count()})$.

\textbf{Complexity:} Constant.

\textbf{b.bucket_size(n)}

\textbf{Result: size_type}
\textbf{Preconditions:} \texttt{n} shall be in the range $[0, \text{b.bucket_count()})$.
\textbf{Returns:} The number of elements in the \texttt{n}th bucket.

\textbf{Complexity:} $O(\text{b.bucket_size(n)})$
b.begin(n)

Result: local_iterator; const_local_iterator for const b.

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

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

Complexity: Constant.

b.end(n)

Result: local_iterator; const_local_iterator for const b.

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

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

Complexity: Constant.

b.cbegin(n)

Result: const_local_iterator

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

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

Complexity: Constant.

b.cend(n)

Result: const_local_iterator

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

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

Complexity: Constant.

b.load_factor()

Result: float

Returns: The average number of elements per bucket.

Complexity: Constant.

b.max_load_factor()

Result: float

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

Complexity: Constant.

a.max_load_factor(z)

Result: void

Preconditions: z is positive. May change the container’s maximum load factor, using z as a hint.

Complexity: Constant.

a.rehash(n)

Result: void

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

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

a.reserve(n)

Effects: Equivalent to a.rehash(ceil(n / a.max_load_factor())).
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 \(\Theta(N^2)\), 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 \(Pred\).

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

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

The \(insert\) and \(emplace\) members shall not affect the validity of iterators if \((N+n) <= z \times B\), 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.

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.

The member function templates \(find, count, equal_range, contains, extract,\) and \(erase\) shall not participate in overload resolution unless the \(qualified-ids\) \(Pred::is_transparent\) 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.

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

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).

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.

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.

\(\S\) 22.2.8.2

848
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]

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>30.4</td>
<td>Constants  &lt;regex&gt;</td>
</tr>
<tr>
<td>30.5</td>
<td>Exception type</td>
</tr>
<tr>
<td>30.6</td>
<td>Traits</td>
</tr>
<tr>
<td>30.7</td>
<td>Regular expression template</td>
</tr>
<tr>
<td>30.8</td>
<td>Submatches</td>
</tr>
<tr>
<td>30.9</td>
<td>Match results</td>
</tr>
<tr>
<td>30.10</td>
<td>Algorithms</td>
</tr>
<tr>
<td>30.11</td>
<td>Iterators</td>
</tr>
<tr>
<td>30.12</td>
<td>Grammar</td>
</tr>
</tbody>
</table>

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-names 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,
   —  X denotes a traits class defining types and functions for the character container type charT;
   —  u is an object of type X;
   —  v is an object of type const X;
   —  p is a value of type const charT*;
   —  I1 and I2 are input iterators (23.3.5.3);
   —  F1 and F2 are forward iterators (23.3.5.5);
   —  c is a value of type const charT;
   —  s is an object of type X::string_type;
   —  cs is an object of type const X::string_type;
   —  b is a value of type bool;
   —  I is a value of type int;
   —  cl is an object of type X::char_class_type; and

§ 30.2
loc is an object of type \( X::\text{locale\_type} \).

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

\[\text{typename } X::\text{char\_type}\]

\[\text{Return type: } \text{charT}, \text{ the character container type used in the implementation of class template } \text{basic\_regex}.\]

\[\text{typename } X::\text{string\_type}\]

\[\text{Return type: } \text{basic\_string}\angle \text{charT}\rangle\]

\[\text{typename } X::\text{locale\_type}\]

\[\text{Return type: } \text{A copy constructible type that represents the locale used by the traits class.}\]

\[\text{typename } X::\text{char\_class\_type}\]

\[\text{Return type: } \text{A bitmask type (16.3.3.3.4) representing a particular character classification.}\]

\[X::\text{length(p)}\]

\[\text{Return type: } \text{size\_t}\]

\[\text{Returns: The smallest } i \text{ such that } p[i] = 0.\]

\[\text{Complexity: Linear in } i.\]

\[v.\text{translate(c)}\]

\[\text{Return type: } X::\text{char\_type}\]

\[\text{Returns: A character such that for any character } d \text{ that is to be considered equivalent to } c \text{ then } v.\text{translate}(c) = v.\text{translate}(d).\]

\[v.\text{translate\_nocase(c)}\]

\[\text{Return type: } X::\text{char\_type}\]

\[\text{Returns: For all characters } C \text{ that are to be considered equivalent to } c \text{ when comparisons are to be performed without regard to case, then } v.\text{translate\_nocase}(c) = v.\text{translate\_nocase}(C).\]

\[v.\text{transform(F1, F2)}\]

\[\text{Return type: } X::\text{string\_type}\]

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

\[v.\text{transform\_primary(F1, F2)}\]

\[\text{Return type: } X::\text{string\_type}\]

\[\text{Returns: A sort key for the character sequence designated by the iterator range } [F1, F2) \text{ such that if the character sequence } [G1, G2) \text{ sorts before the character sequence } [H1, H2) \text{ when character case is not considered then } v.\text{transform\_primary}(G1, G2) < v.\text{transform\_primary}(H1, H2).\]

\[v.\text{lookup\_collatename(F1, F2)}\]

\[\text{Return type: } X::\text{string\_type}\]

\[\text{Returns: A sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range } [F1, F2). \text{ Returns an empty string if the character sequence is not a valid collating element.}\]

\[v.\text{lookup\_classname(F1, F2, b)}\]

\[\text{Return type: } X::\text{char\_class\_type}\]

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

```cpp
v.isctype(c, cl)
```

*Return type:* `bool`

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

```cpp
v.value(c, I)
```

*Return type:* `int`

*Returns:* Returns the value represented by the digit \( c \) in base \( I \) if the character \( c \) is a valid digit in base \( I \); otherwise returns -1.

[Note 2: The value of \( I \) will only be 8, 10, or 16. — end note]

```cpp
c.u.imbue(loc)
```

*Return type:* `\( X::locale_type \)`

*Effects:* Imbues \( u \) with the locale \( loc \) and returns the previous locale used by \( u \) if any.

```cpp
c.v.getloc()
```

*Return type:* `\( X::locale_type \)`

*Returns:* Returns the current locale used by \( v \), if any.

[Note 3: Class template `regex_traits` meets the requirements for a regular expression traits class when it is specialized for `char` or `wchar_t`. This class template is described in the header `<regex>`, and is described in 30.6. — end note]

### 30.3 Header `<regex>` synopsis

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

namespace std {

    // 30.4, regex constants
    namespace regex_constants {
        using syntax_option_type = T1;
        using match_flag_type = T2;
        using error_type = T3;
    }

    // 30.5, class regex_error
    class regex_error;

    // 30.6, class template regex_traits
    template<class charT> struct regex_traits;

    // 30.7, class template basic_regex
    template<class charT, class traits = regex_traits<charT>> class basic_regex;

    using regex = basic_regex<char>;
    using wregex = basic_regex<wchar_t>;

    // 30.7.6, basic_regex swap
    template<class charT, class traits>
        void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2);

    // 30.8, class template sub_match
    template<class BidirectionalIterator>
        class sub_match;

    using csub_match = sub_match<const char*>;
    using wcsub_match = sub_match<const wchar_t*>;
```