Concepts for the C++0x Standard Library: Containers (Revision 1)

Douglas Gregor, Indiana University ⟨dgregor@osl.iu.edu⟩
Pablo Halpern, Bloomberg, L.P. ⟨phalpern@halpernwightsoftware.com⟩

Document number: N2623=08-0133
Revises document number: N2085=06-0155
Date: 2008-05-19
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Introduction

This document proposes changes to Chapter 23 of the C++ Standard Library in order to make full use of concepts [1]. We make every attempt to provide complete backward compatibility with the pre-concept Standard Library, and note each place where we have knowingly changed semantics.

This document is formatted in the same manner as the latest working draft of the C++ standard (N2588). Future versions of this document will track the working draft and the concepts proposal as they evolve. Wherever the numbering of a (sub)section matches a section of the working paper, the text in this document should be considered replacement text, unless editorial comments state otherwise. All editorial comments will have a gray background. Changes to the replacement text are categorized and typeset as additions, ~~removals~~, or ~~changes~~modifications.

Chapter 23 Containers library [containers]

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

2 The following subclauses describe container ~~requirements~~concepts, and components for sequences and associative containers, as summarized in Table 1:

Table 1: Containers library summary

Subclause	Header(s)
23.1 Requirements	
23.1.1 Concepts	`<container_concepts>`

23.2 Sequences	`<array>`
	`<deque>`
	`<list>`
	`<queue>`
	`<stack>`
	`<vector>`

23.3 Associative containers	`<map>` `<set>`
[template.bitset] `bitset`	`<bitset>`
23.4 Unordered associative containers	`<unordered_map>` `<unordered_set>`

23.1 Container requirements [container.requirements]

Unlike with other sections containing requirements tables, we have opted not to completely replace everything in 23.1 with a set of concepts. This decision is due to the unique nature of the container requirements, which don't really correspond to concepts because they aren't used in algorithms. Rather, the requirements tables in 23.1 are shorthand descriptions for all of the containers in this section; each container meets some subset of the requirements stated, sometimes with differing requirements on the container's value type for the same operation. Those container concepts that are actually needed (e.g., for the container adaptors 23.2.5) are specified in the new section 23.1.1.

3 Objects stored in these components shall be constructed using `construct_element` ([construct.element]) and destroyed using the `destroy` member function of the container's allocator ([allocator.concepts]). A container may directly call constructors and destructors for its stored objects, without calling the `construct_element` or `destruct` functions, if the allocator models the `MinimalAllocator` concept. ~~For each operation that inserts an element of type `T` into a container (`insert`, `push_back`, `push_front`, `emplace`, etc.) with arguments `args...` `T` shall be `ConstructibleAsElement`,~~

~~as described in table [constructibleaselement].~~ [*Note:*If the component is instantiated with a scoped allocator of type `A` (i.e., an allocator ~~for which `is_scoped_allocator<A>::value` is true~~that meets the requirements of the `ScopedAllocator` concept), then `construct_element` may pass an inner allocator argument to `T`'s constructor. *— end note*]

4 ~~In table [constructibleaselement], `T` denotes an object type, `A` denotes an allocator, `I` denotes an allocator of type `A::inner_allocator_type` (if any),and `Args` denotes a template parameter pack~~

Remove Table 88: `ConstructibleAsElement<A, T, Args>` requirements

9 Copy and move constructors for all container types defined in this clause obtain an allocator by calling ~~`allocator_propogation_map`~~`AllocatorPropagation<allocator_type>::select_for_copy_construction()` on their respective first parameters. All other constructors for these container types take an `Allocator&` argument ([allocator.requirements]), an allocator whose value type is the same as the container's value type. A copy of this argument is used for any memory allocation 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 calling ~~`allocator_propogation_map`~~`AllocatorPropagation<allocator_type>::move_assign()`, ~~`allocator_propogation_map`~~`AllocatorPropagation<allocator_type>::copy_assign()`, or ~~`allocator_propogation_map`~~`AllocatorPropagationallocator_type>::swap()` 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` object used to construct the container, or to replace the allocator.

Add the following new section [container.concepts]

23.1.1 Container concepts [container.concepts]

1 The `container_concepts` header describes requirements on the template arguments used in container adaptors.

Header `<container_concepts>` synopsis

```
namespace std {
  auto concept Container<typename C> see below
  auto concept SequenceContainer<typename C> see below
  auto concept FrontInsertionSequence<typename C> see below
  auto concept BackInsertionSequence<typename C> see below
  auto concept InsertionSequence<typename C> see below
}
```

```
auto concept Container<typename C> {
  ObjectType           value_type      = typename C::value_type;
  typename             reference       = typename C::reference;
  typename             const_reference = typename C::const_reference;
  UnsignedIntegralLike size_type       = typename C::size_type;

  ForwardIterator iterator;
  ForwardIterator const_iterator;
  requires SameType<ForwardIterator<iterator>::value_type, value_type>
        && SameType<ForwardIterator<const_iterator>::value_type, value_type>;

  bool      C::empty() const;
  size_type C::size() const;
```

```
  iterator       C::begin();
  const_iterator C::begin() const;
  iterator       C::end();
  const_iterator C::end() const;

  void C::swap(C&&);

  axiom ContainerSize(C c) {
    (C.begin() == C.end()) == C.empty();
    (C.begin() != C.end()) == (C.size() > 0);
  }
}
```

2 *Note:* describes a container, which provides iteration through a sequence of elements stored in the container.

3 *Requires:* for a (possibly `const`-qualified) container `C`, `[C.begin(), C.end())` is a valid range.

```
auto concept SequenceContainer<typename C> : Container<C> {
  reference       C::front();
  const_reference C::front() const;
  reference       C::back();
  const_reference C::back() const;

  axiom AccessFront(C c) {
    if (c.begin() != c.end()) c.front() == *c.begin();
  }

  axiom AccessBack(C c) {
    if (c.begin() != c.end()) c.back() == *(--c.end());
  }
}
```

4 *Note:* describes a sequence container, which stores its elements in the order in which they were added.

```
auto concept FrontInsertionSequence<typename C> : SequenceContainer<C> {
  void C::push_front(const value_type&);
  void C::pop_front();

  axiom FrontInsertion(C c, value_type x) {
    c == (c.push_front(x), c.pop_front());
  }
}
```

5 *Note:* describes a container that can be modified by adding or removing elements from the front of the sequence.

```
auto concept BackInsertionSequence<typename C> : SequenceContainer<C> {
  void C::push_back(const value_type&);
  void C::pop_back();

  axiom BackInsertion(C c, value_type x) {
```

```
    c == (c.push_back(x), c.pop_back());
  }
}
```

6 *Note:* describes a container that can be modified by adding or removing elements from the back of the sequence.

```
auto concept InsertionSequence<typename C> : SequenceContainer<C> {
  iterator C::insert(iterator, const value_type&);
}
```

7 *Note:* describes a container that can be modified by inserting elements at any position within the sequence.

23.2 Sequences [sequences]

1 Headers `<array>`, `<deque>`, `<forward_list>`, `<list>`, `<queue>`, `<stack>`, and `<vector>`.

Header `<array>` synopsis

```
namespace std {
  template <classObjectType T, size_t N >
    requires Destructible<T>
    struct array;
  template <classEqualityComparable T, size_t N>
    bool operator==(const array<T,N>& x, const array<T,N>& y);
  template <classEqualityComparable T, size_t N>
    bool operator!=(const array<T,N>& x, const array<T,N>& y);
  template <classLessThanComparable T, size_t N>
    bool operator<(const array<T,N>& x, const array<T,N>& y);
  template <classLessThanComparable T, size_t N>
    bool operator>(const array<T,N>& x, const array<T,N>& y);
  template <classLessThanComparable T, size_t N>
    bool operator<=(const array<T,N>& x, const array<T,N>& y);
  template <classLessThanComparable T, size_t N>
    bool operator>=(const array<T,N>& x, const array<T,N>& y);
  template <classSwappable T, size_t N >
    void swap(array<T,N>& x, array<T,N>& y);

  template <classObjectType T> class tuple_size;
  template <int I, classObjectType T>
    class tuple_element;
  template <classObjectType T, size_t N>
    struct tuple_size<array<T, N> >;
  template <int I, class T, size_t N>
    requires True<(I > 0 && I < N)>
    struct tuple_element<I, array<T, N> >;
  template <int I, class T, size_t N>
    requires True<(I > 0 && I < N)>
    T& get(array<T, N>&);
  template <int I, class T, size_t N>
    requires True<(I > 0 && I < N)>
    const T& get(const array<T, N>&);
```

```
}
```

Header <deque> synopsis

```
namespace std {
  template <classObjectType T, classRandomAccessAllocator Allocator = allocator<T> >
    requires Destructible<T>
    class deque;
  template <classEqualityComparable T, class Allocator>
    bool operator==(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator<(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <classEqualityComparable T, class Allocator>
    bool operator!=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator>(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator>=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator<=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(deque<T,Allocator>&& x, deque<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(deque<T,Allocator>& x, deque<T,Allocator>&& y);
}
```

Header <forward_list> synopsis

```
namespace std {
  template <classObjectType T, classAllocator Allocator = allocator<T> >
    requires Destructible<T>
    class forward_list;
  template <classEqualityComparable T, class Allocator>
    bool operator==(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator< (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <classEqualityComparable T, class Allocator>
    bool operator!=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator> (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator>=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator<=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
}
```

Header <list> synopsis

```
namespace std {
  template <classObjectType T, classAllocator Allocator = allocator<T> >
    requires Destructible<T>
    class list;
  template <classEqualityComparable T, class Allocator>
    bool operator==(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator< (const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <classEqualityComparable T, class Allocator>
    bool operator!=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator> (const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator>=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator<=(const list<T,Allocator>& x, const list<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(list<T,Allocator>& x, list<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(list<T,Allocator>&& x, list<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(list<T,Allocator>& x, list<T,Allocator>&& y);
}
```

Header <queue> synopsis

```
namespace std {
  template <classObjectType T, class Container = deque<T> >
    requires FrontInsertionSequence<Cont> && BackInsertionSequence<Cont>
             && SameType<T, Cont::value_type>
    class queue;
  template <class T, classEqualityComparable Container>
    bool operator==(const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, classLessThanComparable Container>
    bool operator< (const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, classEqualityComparable Container>
    bool operator!=(const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, classLessThanComparable Container>
    bool operator> (const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, classLessThanComparable Container>
    bool operator>=(const queue<T, Container>& x,const queue<T, Container>& y);
  template <class T, classLessThanComparable Container>
    bool operator<=(const queue<T, Container>& x,const queue<T, Container>& y);
  template <classObjectType T, class Allocator>
    void swap(queue<T,Allocator>& x, queue<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(queue<T,Allocator>&& x, queue<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(queue<T,Allocator>& x, queue<T,Allocator>&& y);
```

```
  template <classObjectType T, classBackInsertionSequence Container = vector<T>,
        classPredicate<auto, T, T> Compare = less<typename Container::value_type> >
    requires SameType<Cont::value_type, T> && MutableRandomAccessIterator<Cont::iterator>
             && SwappableIterator<Cont::iterator> && CopyConstructible<Compare>
  class priority_queue;
  template <classObjectType T, class Allocator, Swappable Compare>
    void swap(priority_queue<T,Allocator>& x, priority_queue<T,Allocator>& y);
  template <classObjectType T, class Allocator, Swappable Compare>
    void swap(priority_queue<T,Allocator>&& x, priority_queue<T,Allocator>& y);
  template <classObjectType T, class Allocator, Swappable Compare>
    void swap(priority_queue<T,Allocator>& x, priority_queue<T,Allocator>&& y);
}
```

Header <stack> synopsis

```
namespace std {
  template <classObjectType T, classBackInsertionSequence Container = deque<T> >
    requires SameType<Cont::value_type, T>
    class stack;
  template <classEqualityComparable T, class Container>
    bool operator==(const stack<T, Container>& x,const stack<T, Container>& y);
  template <classLessThanComparable T, class Container>
    bool operator< (const stack<T, Container>& x,const stack<T, Container>& y);
  template <classEqualityComparable T, class Container>
    bool operator!=(const stack<T, Container>& x,const stack<T, Container>& y);
  template <classLessThanComparable T, class Container>
    bool operator> (const stack<T, Container>& x,const stack<T, Container>& y);
  template <classLessThanComparable T, class Container>
    bool operator>=(const stack<T, Container>& x,const stack<T, Container>& y);
  template <classLessThanComparable T, class Container>
    bool operator<=(const stack<T, Container>& x,const stack<T, Container>& y);
  template <classObjectType T, class Allocator>
    void swap(stack<T,Allocator>& x, stack<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(stack<T,Allocator>&& x, stack<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(stack<T,Allocator>& x, stack<T,Allocator>&& y);
}
```

Header <vector> synopsis

```
namespace std {
  template <classObjectType T, classRandomAccessAllocator Allocator = allocator<T> >
    requires Destructible<T>
    class vector;
  template <classEqualityComparable T, class Allocator>
    bool operator==(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator< (const vector<T,Allocator>& x,const vector<T,Allocator>& y);
  template <classEqualityComparable T, class Allocator>
    bool operator!=(const vector<T,Allocator>& x,const vector<T,Allocator>& y);
```

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

template <classRandomAccessAllocator Allocator> class vector<bool,Allocator>;
```

23.2.1 Class template `array` [array]

1 The header `<array>` defines a class template for storing fixed-size sequences of objects. An `array` supports random access iterators. An instance of `array<T, N>` stores `N` elements of type `T`, so that `size() == N` is an invariant. The elements of an `array` are stored contiguously, meaning that if `a` is an `array<T, N>` then it obeys the identity `&a[n] == &a[0] + n` for all `0 <= n < N`.

2 An `array` is an aggregate ([dcl.init.aggr]) that can be initialized with the syntax

```
array a = { initializer-list };
```

where *initializer-list* is a comma separated list of up to `N` elements whose types are convertible to `T`.

3 Unless otherwise specified, all `array` operations are as described in 23.1. Descriptions are provided here only for operations on `array` that are not described in that clause or for operations where there is additional semantic information.

```
namespace std {
  template <classObjectType T, size_t N >
  requires Destructible<T>
  struct array {
    // types:
    typedef T &                                  reference;
    typedef const T &                            const_reference;
    typedef implementation defined               iterator;
    typedef implementation defined               const_iterator;
    typedef size_t                               size_type;
    typedef ptrdiff_t                            difference_type;
    typedef T                                    value_type;
    typedef std::reverse_iterator<iterator>      reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    T       elems[N];           // exposition only

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

    requires CopyAssignable<T> void assign(const T& u);
```

```
    requires Swappable<T> void swap(array<T, N> &);

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

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

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

    // capacity:
    constexpr size_type size() const;
    constexpr size_type max_size() const;
    bool      empty() const;

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

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

4 [*Note:* The member variable `elems` is shown for exposition only, to empahasize that `array` is a class aggregate. The name `elems` is not part of `array`'s interface. *— end note*]

23.2.1.1 `array` constructors, copy, and assignment [array.cons]

1 The conditions for an aggregate ([dcl.init.aggr]) shall be met. Class `array` relies on the implicitly-declared special member functions ([class.ctor], [class.dtor], and [class.copy]) to conform to the container requirements table in 23.1.

23.2.1.2 `array` specialized algorithms [array.special]

```
template <~~class~~Swappable T, size_t N> void swap(array<T,N>& x, array<T,N>& y);
```

1 *Effects:*

```
swap_ranges(x.begin(), x.end(), y.begin() );
```

23.2.1.3 `array::size` [array.size]

~~`template <class T, size_t N>`~~ `size_type` ~~`array<T,N>::`~~`size();`

1 *Returns:* `N`

23.2.1.4 `array::data` [array.data]

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

1 *Returns:* `elems`.

23.2.1.5 Zero sized arrays [array.zero]

1 `array` shall provide support for the special case `N == 0`.

2 In the case that `N == 0`, `begin() == end() ==` unique value. The return value of `data()` is unspecified.

3 The effect of calling `front()` or `back()` for a zero-sized array is implementation defined.

23.2.1.6 Tuple interface to class template `array` [array.tuple]

```
tuple_size<array<T, N> >::value
```

1 *Return type:* integral constant expression.

2 *Value:* `N`

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

3 *Requires:* `0 <= I < N`. The program is ill-formed if `I` is out of bounds.

4 *Value:* The type T.

```
template <int I, class T, size_t N>
  requires True<(I > 0 && I < N)>
  T& get(array<T, N>& a);
```

5 ~~*Requires:* `0 <= I < N`. The program is ill-formed if `I` is out of bounds.~~

Returns: A reference to the `I`th element of `a`, where indexing is zero-based.

```
template <int I, class T, size_t N>
  requires True<(I > 0 && I < N)>
  const T& get(const array<T, N>& a);
```

6 ~~*Requires:* `0 <= I < N`. The program is ill-formed if `I` is out of bounds.~~

7 *Returns:* A const reference to the `I`th element of `a`, where indexing is zero-based.

23.2.2 Class template deque [deque]

1 A deque is a sequence container that, like a vector (23.2.6), supports random access iterators. In addition, it supports constant time insert and erase operations at the beginning or the end; insert and erase in the middle take linear time. That is, a deque is especially optimized for pushing and popping elements at the beginning and end. As with vectors, storage management is handled automatically.

2 A deque satisfies all of the requirements of a container, of a reversible container (given in tables in 23.1), of a sequence container, including the optional sequence container requirements ([sequence.reqmts]), and of an allocator-aware container (Table [tab:containers.allocatoraware]). Descriptions are provided here only for operations on deque that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
  template <classObjectType T, classRandomAccessAllocator Allocator = allocator<T> >
  requires Destructible<T>
  class deque {
  public:
    // types:
    typedef typename Allocator::reference         reference;
    typedef typename Allocator::const_reference   const_reference;
    typedef implementation-defined                iterator;        // See 23.1
    typedef implementation-defined                const_iterator;  // See 23.1
    typedef implementation-defined                size_type;       // See 23.1
    typedef implementation-defined                difference_type;// See 23.1
    typedef T                                     value_type;
    typedef Allocator                             allocator_type;
    typedef typename Allocator::pointer           pointer;
    typedef typename Allocator::const_pointer     const_pointer;
    typedef std::reverse_iterator<iterator>       reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // 23.2.2.1 construct/copy/destroy:
    explicit deque(const Allocator& = Allocator());
    requires DefaultConstructible<T> explicit deque(size_type n);
    requires ConstructibleAsElement<Alloc, T, const T&>
      deque(size_type n, const T& value,const Allocator& = Allocator());
    template <class InputIteratorInputIterator Iter>
      requires ConstructibleAsElement<Alloc, T, Iter::reference>
      deque(InputIteratorIter first, InputIteratorIter last,const Allocator& = Allocator());
    requires ConstructibleAsElement<Alloc, T, const T&> deque(const deque<T,Allocator>& x);
    deque(deque&&);
    requires ConstructibleAsElement<Alloc, T, const T&> deque(const deque&, const Allocator&);
    deque(deque&&, const Allocator&);

   ~deque();
    requires ConstructibleAsElement<Alloc, T, const T&>
      deque<T,Allocator>& operator=(const deque<T,Allocator>& x);
    deque<T,Allocator>& operator=(const deque<T,Allocator>&& x);
    template <class InputIteratorInputIterator Iter>
      requires ConstructibleAsElement<Alloc, T, Iter::reference>
```

```
  void assign(InputIterator first, InputIterator last);
requires ConstructibleAsElement<Alloc, T, const T&>
  void assign(size_type n, const T& t);
allocator_type get_allocator() const;

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

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

// 23.2.2.2 capacity:
size_type size() const;
size_type max_size() const;
requires DefaultConstructible<T> && MoveAssignable<T>
  void resize(size_type sz);
requires ConstructibleAsElement<Alloc, T, const T&> && MoveAssignable<T>
  void resize(size_type sz, const T& c);
bool empty() const;

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

// 23.2.2.3 modifiers:
template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...>
  void push_front(Args&&... args);
template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...>
  void push_back(Args&&... args);

template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...> && MoveAssignable<T>
  iterator emplace(const_iterator position, Args&&... args);
```

```
    requires ConstructibleAsElement<Alloc, T, const T&> && MoveAssignable<T>
      iterator insert(const_iterator position, const T& x);
    requires ConstructibleAsElement<Alloc, T, T&&> && MoveAssignable<T>
      iterator insert(const_iterator position, T&& x);
    requires ConstructibleAsElement<Alloc, T, const T&> && MoveAssignable<T>
      void insert(const_iterator position, size_type n, const T& x);
    template <class InputIteratorInputIterator Iter>
      requires ConstructibleAsElement<Alloc, T, Iter::reference> && MoveAssignable<T>
      void insert(const_iterator position, InputIteratorIter first, InputIteratorIter last);

    void pop_front();
    void pop_back();

    requires MoveAssignable<T> iterator erase(const_iterator position);
    requires MoveAssignable<T> iterator erase(const_iterator first, const_iterator last);
    void     swap(deque<T,Allocator>&&);
    void     clear();
  };

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

  // specialized algorithms:
  template <classObjectType T, class Allocator>
    void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(deque<T,Allocator>&& x, deque<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(deque<T,Allocator>& x, deque<T,Allocator>&& y);

  template <class T, class Alloc
    struct constructible_with_allocator_suffix<deque<T, Alloc> >
      : true_type { };
}
```

23.2.2.1 `deque` constructors, copy, and assignment [deque.cons]

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

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

2 *Complexity:* Constant.

```
requires DefaultConstructible<T> explicit deque(size_type n);
```

3 *Effects:*Constructs a deque with n default constructed elements.

4 @~~*Requires:* T shall be DefaultConstructible.~~@

5 *Complexity:*Linear in *n*.

```
requires ConstructibleAsElement<Alloc, T, const T&>
deque(size_type n, const T& value,
      const AllocatorAlloc& = AllocatorAlloc());
```

6 *Effects:* Constructs a deque with *n* copies of *value*, using the specified allocator.

7 @~~*Requires:* T shall be CopyConstructible.~~@

8 *Complexity:* Linear in *n*.

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, T, Iter::reference>
  deque(InputIteratorIter first, InputIteratorIter last,
        const AllocatorAlloc& = AllocatorAlloc());
```

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

10 *Complexity:* distance(*first*, *last*).

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, T, Iter::reference>
  void assign(InputIteratorIter first, InputIteratorIter last);
```

11 *Effects:*

```
erase(begin(), end());
insert(begin(), first, last);
```

```
requires ConstructibleAsElement<Alloc, T, const T&>
void assign(size_type n, const T& t);
```

12 *Effects:*

```
erase(begin(), end());
insert(begin(), n, t);
```

23.2.2.2 deque capacity [deque.capacity]

```
requires DefaultConstructible<T> && MoveAssignable<T>
  void resize(size_type sz);
```

1 *Effects:*If `sz < size()`, equivalent to `erase(begin() + sz, end());`. If `size() < sz`, appends `sz - size()` default constructed elements to the sequence.

2 ~~*Requires:* T shall be `DefaultConstructible`.~~

```
requires ConstructibleAsElement<Alloc, T, const T&> && MoveAssignable<T>
  void resize(size_type sz, const T& c);
```

3 *Effects:*

```
if (sz > size())
  insert(end(), sz-size(), c);
else if (sz < size())
  erase(begin()+sz, end());
else
  ;                    // do nothing
```

4 ~~*Requires:* T shall be `CopyConstructible`.~~

23.2.2.3 `deque` modifiers [deque.modifiers]

```
requires ConstructibleAsElement<Alloc, T, const T&> && MoveAssignable<T>
  iterator insert(const_iterator position, const T& x);
requires ConstructibleAsElement<Alloc, T, T&&> && MoveAssignable<T>
  iterator insert(const_iterator position, T&& x);
requires ConstructibleAsElement<Alloc, T, const T&> && MoveAssignable<T>
  void     insert(const_iterator position, size_type n, const T& x);
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, T, Iter::reference> && MoveAssignable<T>
  void insert(const_iterator position,
              InputIteratorIter first, InputIteratorIter last);

template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...>
  void push_front(Args&&... args);
template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...>
  void push_back(Args&&... args);
template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...> && MoveAssignable<T>
  iterator emplace(const_iterator position, Args&&... args);
```

1 *Effects:* An insertion in the middle of the deque invalidates all the iterators and references to elements of the deque. An insertion at either end of the deque invalidates all the iterators to the deque, but has no effect on the validity of references to elements of the deque.

2 *Remarks:* If an exception is thrown other than by the copy constructor or assignment operator of `T` there are no effects.

3 *Complexity:* The complexity is linear in the number of elements inserted plus the lesser of the distances to the beginning and end of the deque. Inserting a single element either at the beginning or end of a deque always takes

constant time and causes a single call to a constructor of `T`.

```
requires MoveAssignable<T> iterator erase(const_iterator position);
requires MoveAssignable<T> iterator erase(const_iterator first, const_iterator last);
```

4 *Effects:* An erase in the middle of the deque invalidates all the iterators and references to elements of the deque. An erase at either end of the deque invalidates only the iterators and the references to the erased elements.

5 *Complexity:* The number of calls to the destructor is the same as the number of elements erased, but the number of the calls to the assignment operator is at most equal to the minimum of the number of elements before the erased elements and the number of elements after the erased elements.

6 *Throws:* Nothing unless an exception is thrown by the copy constructor or assignment operator of `T`.

23.2.2.4 `deque` specialized algorithms [deque.special]

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

1 *Effects:*

```
x.swap(y);
```

23.2.3 Class template `forward_list` [forwardlist]

1 A `forward_list` is a container that supports forward iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Fast random access to list elements is not supported. [*Note:*It is intended that `forward_list` have zero space or time overhead relative to a hand-written C-style singly linked list. Features that would conflict with that goal have been omitted. — *end note*]

2 A `forward_list` satisfies all of the requirements of a container (table [tab:containers.container.requirements]), except that the `size()` member function is not provided. Descriptions are provided here only for operations on `forward_list` that are not described in that table or for operations where there is additional semantic information.

```
namespace std {
  template <classObjectType T, classAllocator Allocator = allocator<T> >
  requires Destructible<T>
  class forward_list {
  public:
    // types:
    typedef typename Allocator::reference reference;
    typedef typename Allocator::const_reference const_reference;
    typedef implementation-defined iterator;       // See 23.1
    typedef implementation-defined const_iterator; // See 23.1
    typedef implementation-defined size_type;      // See 23.1
    typedef implementation-defined difference_type;// See 23.1
```

```
typedef T value_type;
typedef Allocator allocator_type;
typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer const_pointer;

// 23.2.3.1 construct/copy/destroy:
explicit forward_list(const Allocator& = Allocator());
requires DefaultConstructible<T> explicit forward_list(size_type n);
requires ConstructibleAsElement<Alloc, T, const T&>
  forward_list(size_type n, const T& value,
               const Allocator& = Allocator());
template <class InputIteratorInputIterator Iter>
  ConstructibleAsElement<Alloc, T, Iter::reference>
  forward_list(InputIteratorIter first, InputIteratorIter last,
               const Allocator& = Allocator());
requires ConstructibleAsElement<Alloc, T, const T&>
  forward_list(const forward_list<T,Allocator>& x);
forward_list(forward_list<T,Allocator>&& x);
~forward_list();
requires ConstructibleAsElement<Alloc, T, const T&>
  forward_list<T,Allocator>& operator=(const forward_list<T,Allocator>& x);
forward_list<T,Allocator>& operator=(forward_list<T,Allocator>&& x);
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, T, Iter::reference>
  void assign(InputIteratorIter first, InputIteratorIter last);
ConstructibleAsElement<Alloc, T, const T&> void assign(size_type n, const T& t);
allocator_type get_allocator() const;

// 23.2.3.2 iterators:
iterator before_begin();
const_iterator before_begin() const;
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

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

// capacity:
bool empty() const;
size_type max_size() const;

// 23.2.3.3 element access:
reference front();
const_reference front() const;

// 23.2.3.4 modifiers:
template <class... Args>
```

```
    requires ConstructibleAsElement<Alloc, T, Args&&...>
    void push_front(Args&&... args);
  void pop_front();

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

  requires ConstructibleAsElement<Alloc, T, const T&>
    void insert_after(const_iterator position, size_type n, const T& x);
  template <class InputIteratorInputIterator Iter>
    requires ConstructibleAsElement<Alloc, T, Iter::reference>
    void insert_after(const_iterator position, InputIteratorIter first, InputIteratorIter last);

  iterator erase_after(const_iterator position);
  iterator erase_after(const_iterator position, iterator last);
  void swap(forward_list<T,Allocator>&&);

  requires DefaultConstructible<T> void resize(size_type sz);
  requires ConstructibleAsElement<Alloc, T, const T&> void resize(size_type sz, value_type c);
  void clear();

  // 23.2.3.5 forward_list operations:
  void splice_after(const_iterator position, forward_list<T,Allocator>&& x);
  void splice_after(const_iterator position, forward_list<T,Allocator>&& x,
                    const_iterator i);
  void splice_after(const_iterator position, forward_list<T,Allocator>&& x,
                    const_iterator first, const_iterator last);

  requires EqualityComparable<T> void remove(const T& value);
  template <classPredicate<auto, T> Predicate> void remove_if(Predicate pred);

  requires EqualityComparable<T> void unique();
  template <classPredicate<auto, T, T> BinaryPredicate>
    void unique(BinaryPredicate binary_pred);

  requires LessThanComparable<T> void merge(forward_list<T,Allocator>&& x);
  template <classPredicate<auto, T, T> Compare>
    void merge(forward_list<T,Allocator>&& x, Compare comp);

  requires LessThanComparable<T> void sort();
  template <classPredicate<auto, T, T> Compare> void sort(Compare comp);

  void reverse();
};
```

```
  // Comparison operators
  template <classEqualityComparable T, class Allocator>
    bool operator==(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator< (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <classEqualityComparable T, class Allocator>
    bool operator!=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator> (const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator>=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);
  template <classLessThanComparable T, class Allocator>
    bool operator<=(const forward_list<T,Allocator>& x, const forward_list<T,Allocator>& y);

  // 23.2.3.6 specialized algorithms:
  template <classObjectType T, class Allocator>
    void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(forward_list<T,Allocator>&& x, forward_list<T,Allocator>& y);
  template <classObjectType T, class Allocator>
    void swap(forward_list<T,Allocator>& x, forward_list<T,Allocator>&& y);
}
```

23.2.3.1 `forward_list` constructors, copy, assignment [forwardlist.cons]

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

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

2 *Complexity:*Constant.

```
requires DefaultConstructible<T> explicit forward_list(size_type n);
```

3 *Effects:*Constructs a `forward_list` object with `n` default constructed elements.

4 @~~*Requires:* T shall be `DefaultConstructible`.~~@

5 *Complexity:*Linear in `n`.

```
requires ConstructibleAsElement<Alloc, T, const T&>
  forward_list(size_type n, const T& value, const Allocator& = Allocator());
```

6 *Effects:*Constructs a `forward_list` object with `n` copies of `value` using the specified allocator.

7 @~~*Requires:* T shall be `CopyConstructible`.~~@

8 *Complexity:*Linear in `n`.

```
template <class InputIteratorInputIterator Iter>
  ConstructibleAsElement<Alloc, T, Iter::reference>
  forward_list(InputIteratorIter first, InputIteratorIter last,
               const Allocator& = Allocator());
```

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

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

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, T, Iter::reference>
  void assign(InputIteratorIter first, InputIteratorIter last);
```

11 *Effects:*`clear(); insert_after(before_begin(), first, last);`

```
ConstructibleAsElement<Alloc, T, const T&> void assign(size_type n, const T\& t);
```

12 *Effects:*`clear(); insert_after(before_begin(), n, t);`

23.2.3.2 `forward_list` iterators [forwardlist.iter]

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

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

23.2.3.3 `forward_list` element access [forwardlist.access]

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

1 *Returns:*`*begin()`

23.2.3.4 `forward_list` modifiers [forwardlist.modifiers]

1 None of the overloads of `insert_after` shall affect the validity of iterators and reference, and `erase_after` shall invalidate only the iterators and references to the erased elements. If an exception is thrown during `insert_after` there shall be no effect. Insertion of `n` elements into a `forward_list` is linear in `n`, and the number of calls to the copy or move constructor of `T` is exactly equal to `n`. Erasing `n` elements from a `forward_list` is linear time in `n` and the number of calls to the destructor of type `T` is exactly equal to `n`.

```
template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...>
  void push_front(Args&&... args);
```

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

```
void pop_front();
```

3 *Effects:*`erase_after(before_begin())`

```
requires ConstructibleAsElement<Alloc, T, const T&>
  iterator insert_after(const_iterator position, const T& x);
```

`requires ConstructibleAsElement<Alloc, T, T&&>`
`  iterator insert_after(const_iterator position, T&& x);`

4 *Requires:* `position` is dereferenceable or equal to `before_begin()`.

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

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

`requires ConstructibleAsElement<Alloc, T, const T&>`
`  void insert_after(const_iterator position, size_type n, const T& x);`

7 *Requires:* `position` is dereferenceable or equal to `before_begin()`.

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

`template <`~~`class InputIterator`~~`InputIterator Iter>`
`  requires ConstructibleAsElement<Alloc, T, Iter::reference>`
`  void insert_after(const_iterator position, `~~`InputIterator`~~`Iter first, `~~`InputIterator`~~`Iter last);`

9 *Requires:* `position` is dereferenceable or equal to `before_begin()`. `first` and `last` are not iterators in `*this`.

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

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

11 *Requires:* `position` is dereferenceable or equal to `before_begin()`.

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

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

13 *Requires:* The iterator following `position` is dereferenceable.

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

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

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

16 *Requires:* All iterators in the range `[position,last)` are dereferenceable.

17 *Effects:* Erases the elements in the range `[position,last)`.

18 *Returns:* `last`

```
requires DefaultConstructible<T> void resize(size_type sz);
requires ConstructibleAsElement<Alloc, T, const T&> void resize(size_type sz, value_type c);
```

19 *Effects:* If `sz < distance(begin(), end())`, erases the last `distance(begin(), end()) - sz` elements from the list. Otherwise, inserts `sz - distance(begin(), end())` elements at the end of the list. For the first signature the inserted elements are default constructed, and for the second signature they are copies of `c`.

`void clear();`

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

23.2.3.5 `forward_list` operations [forwardlist.ops]

`void splice_after(const_iterator position, forward_list<T,Alloc`~~`ator`~~`>&& x);`

1 *Requires:* `position` is dereferenceable or equal to `before_begin()`. `&x != this`.

2 *Effects:* Inserts the contents of `x` before `position`, and `x` becomes empty. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

3 *Throws:* Nothing.

4 *Complexity:* $\mathscr{O}(1)$

`void splice_after(const_iterator position, forward_list<T,Alloc`~~`ator`~~`>&& x, const_iterator i);`

5 *Requires:* `position` is dereferenceable or equal to `before_begin()`. The iterator following `i` is a dereferenceable iterator in `x`.

6 *Effects:* Inserts the element following `i` into `*this`, following `position`, and removes it from `x`. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

7 *Throws:* Nothing.

8 *Complexity:* $\mathscr{O}(1)$

`void splice_after(const_iterator position, forward_list<T,Alloc`~~`ator`~~`>&& x,`
`                  const_iterator first, const_iterator last);`

9 *Requires:* `position` is dereferenceable or equal to `before_begin()`. `(first,last)` is a valid range in `x`, and all iterators in the range `(first,last)` are dereferenceable. `position` is not an iterator in the range `(first, last)`.

10 *Effects:* Inserts elements in the range `(first,last)` after `position` and removes the elements from `x`. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

`requires EqualityComparable<T>` `void remove(const T& value);`
`template <`~~`class`~~`Predicate<auto, T> Pred`~~`icate`~~`> void remove_if(Pred`~~`icate`~~` pred);`

11 *Effects:* Erases all the elements in the list referred by a list iterator `i` for which the following conditions hold: `*i == value` (for `remove()`), `pred(*i)` is true (for `remove_if()`). This operation shall be stable: the relative order of the elements that are not removed is the same as their relative order in the original list.

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

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

```
requires EqualityComparable<T> void unique();
template <classPredicate<auto, T, T> BinaryPredicate>
  void unique(BinaryPredicate pred);
```

14 *Effects:*Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first + 1,last)` for which `*i == *(i-1)` (for the version with no arguments) or `pred(*i, *(i - 1))` (for the version with a predicate argument) holds.

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

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

```
requires LessThanComparable<T> void merge(forward_list<T,Allocator>&& x);
template <classPredicate<auto, T, T> Compare>
  void merge(forward_list<T,Allocator>&& x, Compare comp)
```

17 *Requires:*`comp` defines a strict weak ordering ([alg.sorting]), and `*this` and `x` are both sorted according to this ordering.

18 *Effects:*Merges `x` into `*this`. This operation shall be stable: for equivalent elements in the two lists, the elements from `*this` shall always precede the elements from `x`. `x` is empty after the merge. If an exception is thrown other than by a comparison there are no effects.

19 *Complexity:*At most `size() + x.size() - 1` comparisons.

```
requires LessThanComparable<T> void sort();
template <classPredicate<auto, T, T> Compare> void sort(Compare comp);
```

20 *Requires:*`operator<` (for the version with no arguments) or `comp` (for the version with a comparison argument) defines a strict weak ordering ([alg.sorting]).

21 *Effects:*Sorts the list according to the `operator<` or the `comp` function object. This operation shall be stable: the relative order of the equivalent elements is preserved. If an exception is thrown the order of the elements in `*this` is unspecified.

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

```
void reverse();
```

23 *Effects:*Reverses the order of the elements in the list.

24 *Throws:*Nothing.

25 *Complexity:*Linear time.

23.2.3.6 `forward_list` specialized algorithms [forwardlist.spec]

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

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

1 *Effects:* `x.swap(y)`

23.2.4 Class template `list` [list]

1 A `list` is a sequence container that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Unlike vectors (23.2.6) and deques (23.2.2), fast random access to list elements is not supported, but many algorithms only need sequential access anyway.

2 A `list` satisfies all of the requirements of a container, of a reversible container (given in two tables in 23.1), of a sequence container, including most of the the optional sequence container requirements ([sequence.reqmts]), and of an allocator-aware container (Table [tab:containers.allocatoraware]). The exceptions are the `operator[]` and `at` member functions, which are not provided.[1] Descriptions are provided here only for operations on `list` that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
  template <classObjectType T, classAllocator Allocator = allocator<T> >
  requires Destructible<T>
  class list {
  public:
    // types:
    typedef typename Allocator::reference          reference;
    typedef typename Allocator::const_reference    const_reference;
    typedef implementation-defined                 iterator;       // See 23.1
    typedef implementation-defined                 const_iterator; // See 23.1
    typedef implementation-defined                 size_type;      // See 23.1
    typedef implementation-defined                 difference_type;// See 23.1
    typedef T                                      value_type;
    typedef Allocator                              allocator_type;
    typedef typename Allocator::pointer            pointer;
    typedef typename Allocator::const_pointer      const_pointer;
    typedef std::reverse_iterator<iterator>        reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // 23.2.4.1 construct/copy/destroy:
    explicit list(const Allocator& = Allocator());
    requires DefaultConstructible<T> explicit list(size_type n);
    requires ConstructibleAsElement<Alloc, T, const T&>
      list(size_type n, const T& value, const Allocator& = Allocator());
    template <class InputIteratorInputIterator Iter>
      requires ConstructibleAsElement<Alloc, T, Iter::reference>
      list(InputIteratorIter first, InputIteratorIter last, const Allocator& = Allocator());
    requires ConstructibleAsElement<Alloc, T, const T&> list(const list<T,Allocator>& x);
    list(list&& x);
    requires ConstructibleAsElement<Alloc, T, const T&> list(const list&, const Allocator&);
```

[1] These member functions are only provided by containers whose iterators are random access iterators.

```
list(list&&, const Allocator&);
~list();
requires ConstructibleAsElement<Alloc, T, const T&>
  list<T,Allocator>& operator=(const list<T,Allocator>& x);
list<T,Allocator>& operator=(list<T,Allocator>&& x);
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, T, Iter::reference>
  void assign(InputIteratorIter first, InputIteratorIter last);
requires ConstructibleAsElement<Alloc, T, const T&>
  void assign(size_type n, const T& t);
allocator_type get_allocator() const;

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

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

// 23.2.4.2 capacity:
bool      empty() const;
size_type size() const;
size_type max_size() const;
requires DefaultConstructible<T> void resize(size_type sz);
requires ConstructibleAsElement<Alloc, T, const T&> void resize(size_type sz, const T& c);

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

// 23.2.4.3 modifiers:
template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...>
  void push_front(Args&&... args);
void pop_front();
template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...>
  void push_back(Args&&... args);
void pop_back();
```

```
    template <class... Args>
      requires ConstructibleAsElement<Alloc, T, Args&&...>
      iterator emplace(const_iterator position, Args&&... args);
    requires ConstructibleAsElement<Alloc, T, const T&>
      iterator insert(const_iterator position, const T& x);
    requires ConstructibleAsElement<Alloc, T, T&&>
      iterator insert(const_iterator position, T&& x);
    requires ConstructibleAsElement<Alloc, T, const T&>
      void insert(const_iterator position, size_type n, const T& x);
    template <class InputIteratorInputIterator Iter>
      requires ConstructibleAsElement<Alloc, T, Iter::reference>
      void insert(const_iterator position, InputIteratorIter first,
                  InputIteratorIter last);

    iterator erase(const_iterator position);
    iterator erase(const_iterator position, const_iterator last);
    void     swap(list<T,Allocator>&&);
    void     clear();

    // 23.2.4.4 list operations:
    void splice(const_iterator position, list<T,Allocator>&& x);
    void splice(const_iterator position, list<T,Allocator>&& x, const_iterator i);
    void splice(const_iterator position, list<T,Allocator>&& x,
                const_iterator first, const_iterator last);

    requires EqualityComparable<T> void remove(const T& value);
    template <classPredicate<auto, T> Predicate> void remove_if(Predicate pred);

    requires EqualityComparable<T> void unique();
    template <classPredicate<auto, T, T> BinaryPredicate>
      void unique(BinaryPredicate binary_pred);

    requires LessThanComparable<T> void merge(list<T,Allocator>&& x);
    template <classBinaryPredicate<auto, T, T> Compare>
      void merge(list<T,Allocator>&& x, Compare comp);

    requires LessThanComparable<T> void sort();
    template <classBinaryPredicate<auto, T, T> Compare> void sort(Compare comp);

    void reverse();
  };

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

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

// specialized algorithms:
template <classObjectType T, class Allocator>
  void swap(list<T,Allocator>& x, list<T,Allocator>& y);
template <classObjectType T, class Allocator>
  void swap(list<T,Allocator>&& x, list<T,Allocator>& y);
template <classObjectType T, class Allocator>
  void swap(list<T,Allocator>& x, list<T,Allocator>&& y);

template <class T, class Alloc>
  struct constructible_with_allocator_suffix<list<T, Alloc> >
    : true_type { };
}
```

23.2.4.1 `list` constructors, copy, and assignment [list.cons]

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

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

2 *Complexity:* Constant.

```
requires DefaultConstructible<T> explicit list(size_type n);
```

3 *Effects:*Constructs a `list` with `n` default constructed elements.

4 ~~*Requires:* `T` shall be `DefaultConstructible`.~~

5 *Complexity:*Linear in `n`.

```
requires ConstructibleAsElement<Alloc, T, const T&>
  list(size_type n, const T& value,
              const Allocator& = Allocator());
```

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

7 ~~*Requires:* `T` shall be `CopyConstructible`.~~

8 *Complexity:* Linear in `n`.

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, T, Iter::reference>
  list(InputIteratorIter first, InputIteratorIter last, const Allocator& = Allocator());
```

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

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

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, T, Iter::reference>
```

```
void assign(InputIteratorIter first, InputIteratorIter last);
```

11 *Effects:* Replaces the contents of the list with the range [first, last).

```
erase(begin(), end());
insert(begin(), n, t);
```

```
requires ConstructibleAsElement<Alloc, T, const T&>
  void assign(size_type n, const T& t);
```

12 *Effects:* Replaces the contents of the list with *n* copies of *t*.

23.2.4.2 list capacity [list.capacity]

```
requires DefaultConstructible<T> void resize(size_type sz);
```

1 *Effects:*If sz < size(), equivalent to list<T>::iterator it = begin(); advance(it, sz); erase(it, end());. If size() < sz, appends sz - size() default constructed elements to the sequence.

2 ~~*Requires:* T shall be DefaultConstructible.~~

```
requires ConstructibleAsElement<Alloc, T, const T&> void resize(size_type sz, const T& c);
```

3 *Effects:*

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

4 ~~*Requires:* T shall be CopyConstructible.~~

23.2.4.3 list modifiers [list.modifiers]

```
requires ConstructibleAsElement<Alloc, T, const T&>
  iterator insert(const_iterator position, const T& x);
requires ConstructibleAsElement<Alloc, T, T&&>
  iterator insert(const_iterator position, T&& x);
requires ConstructibleAsElement<Alloc, T, const T&>
  void insert(const_iterator position, size_type n, const T& x);
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, T, Iter::reference>
  void insert(const_iterator position, InputIteratorIter first,
              InputIteratorIter last);
```

```
template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...>
  void push_front(Args&&... args);
template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...>
  void push_back(Args&&... args);
template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...>
  iterator emplace(const_iterator position, Args&&... args);
```

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

2 *Complexity:* Insertion of a single element into a list takes constant time and exactly one call to a constructor of `T`. Insertion of multiple elements into a list is linear in the number of elements inserted, and the number of calls to the copy constructor or move constructor of `T` is exactly equal to the number of elements inserted.

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

void pop_front();
void pop_back();
void clear();
```

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

4 *Throws:* Nothing.

5 *Complexity:* Erasing a single element is a constant time operation with a single call to the destructor of `T`. Erasing a range in a list is linear time in the size of the range and the number of calls to the destructor of type `T` is exactly equal to the size of the range.

23.2.4.4 `list` operations [list.ops]

1 Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for them.[2)]

2 `list` provides three splice operations that destructively move elements from one list to another. The behavior of splice operations is undefined if `get_allocator() != x.get_allocator()`.

```
void splice(const_iterator position, list<T,Allocator>&& x);
```

3 *Requires:* `&x != this`.

4 *Effects:* Inserts the contents of `x` before `position` and `x` becomes empty. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

5 *Throws:* Nothing

6 *Complexity:* Constant time.

[2)] As specified in [allocator.requirements], the requirements in this clause apply only to lists whose allocators compare equal.

```
void splice(const_iterator position, list<T,Alloc~~ator~~>&& x, iterator i);
```

7 *Effects:* Inserts an element pointed to by `i` from list `x` before `position` and removes the element from `x`. The result is unchanged if `position == i` or `position == ++i`. Pointers and references to `*i` continue to refer to this same element but as a member of `*this`. Iterators to `*i` (including `i` itself) continue to refer to the same element, but now behave as iterators into `*this`, not into `x`.

8 *Throws:* Nothing

9 *Requires:* `i` is a valid dereferenceable iterator of `x`.

10 *Complexity:* Constant time.

```
void splice(const_iterator position, list<T,Alloc~~ator~~>&& x, iterator first,
            iterator last);
```

11 *Effects:* Inserts elements in the range `[first,last)` before `position` and removes the elements from `x`.

12 *Requires:* `[first, last)` is a valid range in `x`. The result is undefined if `position` is an iterator in the range `[first,last)`. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

13 *Throws:* Nothing

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

```
requires EqualityComparable<T> void remove(const T& value);
template <~~class~~Predicate<auto, T> Pred~~icate~~> void remove_if(Pred~~icate~~ pred);
```

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

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

17 *Remarks:* Stable.

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

```
requires EqualityComparable<T> void unique();
template <~~class~~Predicate<auto, T, T> BinaryPredicate> void unique(BinaryPredicate binary_pred);
```

19 *Effects:* Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first + 1,last)` for which `*i == *(i-1)` (for the version of `unique` with no arguments) or `pred(*i, *(i - 1))` (for the version of `unique` with a predicate argument) holds.

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

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

```
requires LessThanComparable<T> void merge(list<T,Alloc~~ator~~>&& x);
template <~~class~~Predicate<auto, T, T> Compare> void merge(list<T,Alloc~~ator~~>&& x, Compare comp);
```

22 *Requires:* `comp` shall define a strict weak ordering ([alg.sorting]), and both the list and the argument list shall be sorted according to this ordering.

23 *Effects:* If `(&x == this)` does nothing; otherwise, merges the two sorted ranges `[begin(), end())` and `[x.begin(), x.end())`. The result is a range in which the elements will be sorted in non-decreasing order according to the ordering defined by `comp`; that is, for every iterator `i`, in the range other than the first, the condition `comp(*i, *(i - 1)` will be false.

24 *Remarks:* Stable. If `(&x != this)` the range `[x.begin(), x.end())` is empty after the merge.

25 *Complexity:* At most `size() + x.size() - 1` applications of `comp` if `(&x != this)`; otherwise, no applications of `comp` are performed. If an exception is thrown other than by a comparison there are no effects.

```
void reverse();
```

26 *Effects:* Reverses the order of the elements in the list.

27 *Throws:* Nothing.

28 *Complexity:* Linear time.

```
requires LessThanComparable<T> void sort();
template <classBinaryPredicate<auto, T, T> Compare> void sort(Compare comp);
```

29 *Requires:* `operator<` (for the first version) or `comp` (for the second version) shall define a strict weak ordering ([alg.sorting]).

30 *Effects:* Sorts the list according to the `operator<` or a `Compare` function object.

31 *Remarks:* Stable.

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

23.2.4.5 `list` specialized algorithms [list.special]

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

1 *Effects:*

```
x.swap(y);
```

23.2.5 Container adaptors [container.adaptors]

1 The container adaptors each take a `Container` template parameter, and each constructor takes a `Container` reference argument. This container is copied into the `Container` member of each adaptor. If the container takes an allocator, then a compatible allocator may be passed in to the adaptor's constructor. Otherwise, normal copy or move construction

is used for the container argument. [*Note:*it is not necessary for an implementation to distinguish between the one-argument constructor that takes a `Container` and the one-argument constructor that takes an `allocator_type`. Both forms use their argument to construct an instance of the container. — *end note*]

23.2.5.1 Class template `queue` [queue]

1 Any sequence container ~~supporting operations `front()`, `back()`, `push_back()` and `pop_front()`~~meeting the requirements of the `FrontInsertionSequence` and `BackInsertionSequence` concepts can be used to instantiate `queue`. In particular, `list` (23.2.4) and `deque` (23.2.2) can be used.

23.2.5.1.1 `queue` definition [queue.defn]

```
namespace std {
  template <classObjectType T, class Container = deque<T> >
  requires FrontInsertionSequence<Cont> && BackInsertionSequence<Cont>
           && SameType<T, Cont::value_type>
  class queue {
  public:
    typedef typename Container::value_type        value_type;
    typedef typename Container::reference         reference;
    typedef typename Container::const_reference   const_reference;
    typedef typename Container::size_type         size_type;
    typedef          Container                    container_type;
  protected:
    Container c;

  public:
    requires CopyConstructible<Cont> explicit queue(const Container&);
    requires MoveConstructible<Cont> explicit queue(Container&& = Container());
    requires MoveConstructible<Cont> queue(queue&& q) : c(std::move(q.c)) {}
    template <class Alloc>
      requires HasConstructor<Cont, Alloc>
      explicit queue(const Alloc&);
    template <class Alloc>
      requires HasConstructor<Cont, Cont, Alloc>
      queue(const Container&, const Alloc&);
    template <class Alloc>
      requires HasConstructor<Cont, Cont&&, Alloc>
      queue(Container&&, const Alloc&);
    template <class Alloc>
      requires HasConstructor<Cont, Cont&&, Alloc&&>
      queue(queue&&, const Alloc&);
    requires MoveAssignable<Cont> queue& operator=(queue&& q)
                                  { c = std::move(q.c); return *this; }

    bool              empty() const     { return c.empty(); }
    size_type         size()  const     { return c.size(); }
    reference         front()           { return c.front(); }
    const_reference   front() const     { return c.front(); }
    reference         back()            { return c.back(); }
    const_reference   back() const      { return c.back(); }
```

```
    void push(const value_type& x)      { c.push_back(x); }
    void push(value_type&& x)           { c.push_back(std::move(x)); }
    void pop()                          { c.pop_front(); }
    void swap(queue&& q)                { c.swap(q.c); }
  };

  template <class T, classEqualityComparable Container>
    bool operator==(const queue<T, Container>& x, const queue<T, Container>& y);
  template <class T, classLessThanComparable Container>
    bool operator< (const queue<T, Container>& x, const queue<T, Container>& y);
  template <class T, classEqualityComparable Container>
    bool operator!=(const queue<T, Container>& x, const queue<T, Container>& y);
  template <class T, classLessThanComparable Container>
    bool operator> (const queue<T, Container>& x, const queue<T, Container>& y);
  template <class T, classLessThanComparable Container>
    bool operator>=(const queue<T, Container>& x, const queue<T, Container>& y);
  template <class T, classLessThanComparable Container>
    bool operator<=(const queue<T, Container>& x, const queue<T, Container>& y);

  template <classObjectType T, class Container>
    void swap(queue<T, Container>& x, queue<T, Container>& y);
  template <classObjectType T, class Container>
    void swap(queue<T, Container>&& x, queue<T, Container>& y);
  template <classObjectType T, class Container>
    void swap(queue<T, Container>& x, queue<T, Container>&& y);

  template <class T, class Cont, class Alloc>
    requires UsesAllocator<Cont, Alloc>
    concept_map UsesAllocator<queue<T, Cont>, Alloc> { }

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

  template <class T, class Container>
    struct constructible_with_allocator_suffix<queue<T, Container> >
      : true_type { };
}
```

23.2.5.1.2 queue operators [queue.ops]

```
template <class T, classEqualityComparable Container>
    bool operator==(const queue<T, Container>& x,
                    const queue<T, Container>& y);
```

1 *Returns:* `x.c == y.c`.

```
template <class T, classEqualityComparable Container>
    bool operator!=(const queue<T, Container>& x,
                    const queue<T, Container>& y);
```

2 *Returns:* `x.c != y.c`.

```
template <class T, classLessThanComparable Container>
    bool operator< (const queue<T, Container>& x,
                    const queue<T, Container>& y);
```

3 *Returns:* `x.c < y.c`.

```
template <class T, classLessThanComparable Container>
    bool operator<=(const queue<T, Container>& x,
                    const queue<T, Container>& y);
```

4 *Returns:* `x.c <= y.c`.

```
template <class T, classLessThanComparable Container>
    bool operator> (const queue<T, Container>& x,
                    const queue<T, Container>& y);
```

5 *Returns:* `x.c > y.c`.

```
template <class T, classLessThanComparable Container>
    bool operator>=(const queue<T, Container>& x,
                    const queue<T, Container>& y);
```

6 *Returns:* `x.c >= y.c`.

23.2.5.1.3 `queue` specialized algorithms [queue.special]

```
template <classObjectType T, class Container>
  void swap(queue<T, Container>& x, queue<T, Container>& y);
template <classObjectType T, class Container>
  void swap(queue<T, Container>&& x, queue<T, Container>& y);
template <classObjectType T, class Container>
  void swap(queue<T, Container>& x, queue<T, Container>&& y);
```

1 *Effects:*`x.swap(y)`.

23.2.5.2 Class template `priority_queue` [priority.queue]

1 Any sequence container with random access iterator and ~~supporting operations `front()`, `push_back()` and `pop_back()`~~that meets the requirements of the `BackInsertionSequence` concept can be used to instantiate `priority_queue`. In particular, `vector` (23.2.6) and `deque` (23.2.2) can be used. Instantiating `priority_queue` also involves supplying a function or function object for making priority comparisons; the library assumes that the function or function object defines a strict weak ordering ([alg.sorting]).

```
namespace std {
  template <classObjectType T, classBackInsertionSequence Container = vector<T>,
        classPredicate<auto, T, T> Compare = less<typename Container::value_type> >
  requires SameType<Cont::value_type, T> && MutableRandomAccessIterator<Cont::iterator>
           && SwappableIterator<Cont::iterator> && CopyConstructible<Compare>
  class priority_queue {
```

```
public:
  typedef typename Container::value_type           value_type;
  typedef typename Container::reference            reference;
  typedef typename Container::const_reference      const_reference;
  typedef typename Container::size_type            size_type;
  typedef          Container                       container_type;
protected:
  Container c;
  Compare comp;

public:
  requires CopyConstructible<Cont> priority_queue(const Compare& x, const Container&);
  requires MoveConstructible<Cont>
    explicit priority_queue(const Compare& x = Compare(), Container&& = Container());
  template <class InputIteratorInputIterator Iter>
    CopyConstructible<Cont> && HasConstructor<Cont, Iter, Iter>
    priority_queue(InputIteratorIter first, InputIteratorIter last,
           const Compare& x, const Container&);
  template <class InputIteratorInputIterator Iter>
    MoveConstructible<Cont> && HasConstructor<Cont, Iter, Iter>
    priority_queue(InputIteratorIter first, InputIteratorIter last,
           const Compare& x = Compare(), Container&& = Container());
  requires MoveConstructible<Cont> priority_queue(priority_queue&&);
  requires MoveAssignable<Cont> priority_queue& operator=(priority_queue&&);
  template <class Alloc>
    requires HasConstructor<Cont, Alloc>
    explicit priority_queue(const Alloc&);
  template <class Alloc>
    requires HasConstructor<Cont, Alloc>
    priority_queue(const Compare&, const Alloc&);
  template <class Alloc>
    requires HasConstructor<Cont, Cont, Alloc>
    priority_queue(const Compare&, const Container&, const Alloc&);
  template <class Alloc>
    requires HasConstructor<Cont, Cont&&, Alloc>
    priority_queue(const Compare&, Container&&, const Alloc&);
  template <class Alloc>
    requires HasConstructor<Cont, Cont&&, Alloc>
    priority_queue(priority_queue&&, const Alloc&);

  bool      empty() const       { return c.empty(); }
  size_type size()  const       { return c.size(); }
  const_reference   top() const { return c.front(); }
  void push(const value_type& x);
  void push(value_type&& x);
  void pop();
  void swap(priority_queue&&);
};
              // no equality is provided
template <classObjectType T, class Container, Swappable Compare>
```

```
    void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>& y);
  template <classObjectType T, class Container, Swappable Compare>
    void swap(priority_queue<T, Container, Compare>&& x, priority_queue<T, Container, Compare>& y);
  template <classObjectType T, class Container, Swappable Compare>
    void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>&& y);

  template <class T, class Cont, class Compare, class Alloc>
    requires UsesAllocator<Cont, Alloc>
    concept_map UsesAllocator<priority_queue<T, Cont, Compare>, Alloc> { }

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

  template <class T, class Container, class Compare>
    struct constructible_with_allocator_suffix<
      priority_queue<T, Container, Compare> >
      : true_type { };
}
```

23.2.5.2.1 `priority_queue` constructors [priqueue.cons]

```
requires CopyConstructible<Cont> priority_queue(const Compare& x, const Container& y);

requires MoveConstructible<Cont>
explicit priority_queue(const Compare& x = Compare(), Container&& y = Container());
```

1 *Requires:* x shall define a strict weak ordering ([alg.sorting]).

2 *Effects:* Initializes `comp` with `x` and `c` with `y` (copy constructing or move constructing as appropriate); calls `make_heap(c.begin(), c.end(), comp)`.

```
template <class InputIteratorInputIterator Iter>
  CopyConstructible<Cont> && HasConstructor<Cont, Iter, Iter>
  priority_queue(InputIteratorIter first, InputIteratorIter last,
         const Compare& x, const Container&);
template <class InputIteratorInputIterator Iter>
  MoveConstructible<Cont> && HasConstructor<Cont, Iter, Iter>
  priority_queue(InputIteratorIter first, InputIteratorIter last,
         const Compare& x = Compare(), Container&& = Container());
```

3 *Requires:* x shall define a strict weak ordering ([alg.sorting]).

4 *Effects:* Initializes `comp` with `x` and `c` with `y` (copy constructing or move constructing as appropriate); calls `c.insert(c.end(), first, last)`; and finally calls `make_heap(c.begin(), c.end(), comp)`.

23.2.5.2.2 `priority_queue` members [priqueue.members]

```
void push(const value_type& x);
```

1 *Effects:*

```
    c.push_back(x);
    push_heap(c.begin(), c.end(), comp);
```

```
void push(value_type&& x);
```

2 *Effects:*

```
    c.push_back(std::move(x));
    push_heap(c.begin(), c.end(), comp);
```

```
void pop();
```

3 *Effects:*

```
    pop_heap(c.begin(), c.end(), comp);
    c.pop_back();
```

23.2.5.2.3 `priority_queue` specialized algorithms [priqueue.special]

```
template <class T, class Container, Compare>
  void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>& y);
template <class T, class Container, Compare>
  void swap(priority_queue<T, Container, Compare>&& x, priority_queue<T, Container, Compare>& y);
template <class T, class Container, Compare>
  void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>&& y);
```

1 *Effects:* `x.swap(y)`.

23.2.5.3 Class template `stack` [stack]

1 Any sequence container ~~supporting operations `back()`, `push_back()` and `pop_back()`~~that meets the requirements of the `BackInsertionSequence` concept can be used to instantiate `stack`. In particular, `vector` (23.2.6), `list` (23.2.4) and `deque` (23.2.2) can be used.

23.2.5.3.1 `stack` definition [stack.defn]

```
  namespace std {
    template <classObjectType T, classBackInsertionSequence Container = deque<T> >
    requires SameType<Cont::value_type, T>
    class stack {
    public:
      typedef typename Container::value_type            value_type;
      typedef typename Container::reference             reference;
      typedef typename Container::const_reference       const_reference;
      typedef typename Container::size_type             size_type;
      typedef          Container                        container_type;
    protected:
      Container c;

    public:
```

```
  requires CopyConstructible<Cont> explicit stack(const Container&);
  requires MoveConstructible<Cont> explicit stack(Container&& = Container());
  template <class Alloc>
    HasConstructor<Cont, Alloc>
    explicit stack(const Alloc&);
  template <class Alloc>
    HasConstructor<Cont, Cont, Alloc>
    stack(const Container&, const Alloc&);
  template <class Alloc>
    HasConstructor<Cont, Cont&&, Alloc>
    stack(Container&&, const Alloc&);
  template <class Alloc>
    HasConstructor<Cont, Cont&&, Alloc>
    stack(stack&&, const Alloc&);

  bool      empty() const             { return c.empty(); }
  size_type size()  const             { return c.size(); }
  reference         top()             { return c.back(); }
  const_reference   top() const       { return c.back(); }
  void push(const value_type& x)      { c.push_back(x); }
  void push(value_type&& x)           { c.push_back(std::move(x)); }
  void pop()                          { c.pop_back(); }
  void swap(stack&& s)                { c.swap(s.c); }
};

template <classEqualityComparable T, class Container>
  bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);
template <classLessThanComparable T, class Container>
  bool operator< (const stack<T, Container>& x, const stack<T, Container>& y);
template <classEqualityComparable T, class Container>
  bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);
template <classLessThanComparable T, class Container>
  bool operator> (const stack<T, Container>& x, const stack<T, Container>& y);
template <classLessThanComparable T, class Container>
  bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);
template <classLessThanComparable T, class Container>
  bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y);
template <classObjectType T, class Allocator>
  void swap(stack<T,Allocator>& x, stack<T,Allocator>& y);
template <classObjectType T, class Allocator>
  void swap(stack<T,Allocator>&& x, stack<T,Allocator>& y);
template <classObjectType T, class Allocator>
  void swap(stack<T,Allocator>& x, stack<T,Allocator>&& y);

template <class T, class Cont, class Alloc>
  requires UsesAllocator<Cont, Alloc>
  concept_map UsesAllocator<stack<T, Cont>, Alloc> { }

template <class T, class Container, class Alloc>
  struct uses_allocator<stack<T, Container>, Alloc>
```

```
      : uses_allocator<Container, Alloc>::type { };

    template <class T, class Container>
      struct constructible_with_allocator_suffix<stack<T, Container> >
        : true_type { };
  }
```

23.2.5.3.2 `stack` operators [stack.ops]

```
template <classEqualityComparable T, class Container>
    bool operator==(const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

1 *Returns:* `x.c == y.c`.

```
template <classEqualityComparable T, class Container>
    bool operator!=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

2 *Returns:* `x.c != y.c`.

```
template <classLessThanComparable T, class Container>
    bool operator< (const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

3 *Returns:* `x.c < y.c`.

```
template <classLessThanComparable T, class Container>
    bool operator<=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

4 *Returns:* `x.c <= y.c`.

```
template <classLessThanComparable T, class Container>
    bool operator> (const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

5 *Returns:* `x.c > y.c`.

```
template <classLessThanComparable T, class Container>
    bool operator>=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

6 *Returns:* `x.c >= y.c`.

23.2.5.3.3 `stack` specialized algorithms [stack.special]

```
template <classObjectType T, class Container>
  void swap(stack<T, Container>& x, stack<T, Container>& y);
template <classObjectType T, class Container>
  void swap(stack<T, Container>&& x, stack<T, Container>& y);
template <classObjectType T, class Container>
```

```
void swap(stack<T, Container>& x, stack<T, Container>&& y);
```

1 *Effects:* `x.swap(y)`.

23.2.6 Class template `vector` [vector]

1 A `vector` is a sequence container that supports random access iterators. In addition, it supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though hints can be given to improve efficiency. The elements of a vector are stored contiguously, meaning that if `v` is a `vector<T, Alloc@`~~`ator`~~`@>` where `T` is some type other than `bool`, then it obeys the identity `&v[n] == &v[0] + n` for all `0 <= n < v.size()`.

2 A `vector` satisfies all of the requirements of a container and of a reversible container (given in two tables in 23.1), of a sequence container, including most of the optional sequence container requirements ([sequence.reqmts]), and of an allocator-aware container (Table [tab:containers.allocatoraware]). The exceptions are the `push_front` and `pop_front` member functions, which are not provided. Descriptions are provided here only for operations on `vector` that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
  template <classObjectType T, classRandomAccessAllocator Allocator = allocator<T> >
  requires Destructible<T>
  class vector {
  public:
    // types:
    typedef typename Allocator::reference          reference;
    typedef typename Allocator::const_reference    const_reference;
    typedef implementation-defined                 iterator;        // See 23.1
    typedef implementation-defined                 const_iterator;  // See 23.1
    typedef implementation-defined                 size_type;       // See 23.1
    typedef implementation-defined                 difference_type;// See 23.1
    typedef T                                      value_type;
    typedef Allocator                              allocator_type;
    typedef typename Allocator::pointer            pointer;
    typedef typename Allocator::const_pointer      const_pointer;
    typedef std::reverse_iterator<iterator>        reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // 23.2.6.1 construct/copy/destroy:
    explicit vector(const Allocator& = Allocator());
    requires DefaultConstructible<T>
      explicit vector(size_type n);
    requires ConstructibleAsElement<Alloc, T, const T&>
      vector(size_type n, const T& value, const Allocator& = Allocator());
    template <class InputIteratorInputIterator Iter>
      requires ConstructibleAsElement<Alloc, T, Iter::reference> && ConstructibleAsElement<Alloc, T,
T&&>
      vector(InputIteratorIter first, InputIteratorIter last,
             const Allocator& = Allocator());
    requires ConstructibleAsElement<Alloc, T, const T&> vector(const vector<T,Allocator>& x);
```

```
vector(vector&&);
requires ConstructibleAsElement<Alloc, T, const T&> vector(const vector&, const Allocator&);
vector(vector&&, const Allocator&);
~vector();
requires ConstructibleAsElement<Alloc, T, const T&>
  vector<T,Allocator>& operator=(const vector<T,Allocator>& x);
vector<T,Allocator>& operator=(vector<T,Allocator>&& x);
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, T, Iter::reference>
  void assign(InputIteratorIter first, InputIteratorIter last);
requires ConstructibleAsElement<Alloc, T, const T&>
  void assign(size_type n, const T& u);
allocator_type get_allocator() const;

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

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

// 23.2.6.2 capacity:
size_type size() const;
size_type max_size() const;
requires DefaultConstructible<T> && ConstructibleAsElement<Alloc, T, T&&>
  void resize(size_type sz);
requires ConstructibleAsElement<Alloc, T, const T&> void resize(size_type sz, const T& c);
size_type capacity() const;
bool      empty() const;
requires ConstructibleAsElement<Alloc, T, T&&> void reserve(size_type n);

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

// 23.2.6.3 data access
```

```
  pointer         data();
  const_pointer   data() const;

  // 23.2.6.4 modifiers:
  template <class... Args>
    requires ConstructibleAsElement<Alloc, T, Args&&...> && ConstructibleAsElement<Alloc, T, T&&>
    void push_back(Args&&... args);
  void pop_back();

  template <class... Args>
    requires ConstructibleAsElement<Alloc, T, Args&&...> && ConstructibleAsElement<Alloc, T, T&&>
    iterator emplace(const_iterator position, Args&&... args);
  requires ConstructibleAsElement<Alloc, T, const T&> && MoveAssignable<T>
    iterator insert(const_iterator position, const T& x);
  requires ConstructibleAsElement<Alloc, T, T&&> && MoveAssignable<T>
    void insert(const_iterator position, T&& x);
  requires ConstructibleAsElement<Alloc, T, const T&> && MoveAssignable<T>
    void insert(const_iterator position, size_type n, const T& x);
  template <class InputIteratorInputIterator Iter>
      requires ConstructibleAsElement<Alloc, T, Iter::reference>
            && ConstructibleAsElement<Alloc, T, T&&> && MoveAssignable<T>
      void insert(const_iterator position,
                  InputIteratorIter first, InputIteratorIter last);
  requires MoveAssignable<T> iterator erase(const_iterator position);
  requires MoveAssignable<T> iterator erase(const_iterator first, const_iterator last);
  void     swap(vector<T,Allocator>&&);
  void     clear();
};

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

// specialized algorithms:
template <classObjectType T, class Allocator>
  void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);
template <classObjectType T, class Allocator>
  void swap(vector<T,Allocator>&& x, vector<T,Allocator>& y);
template <classObjectType T, class Allocator>
  void swap(vector<T,Allocator>& x, vector<T,Allocator>&& y);
```

```
  template <class T, class Alloc>
    struct constructible_with_allocator_suffix<vector<T, Alloc> >
      : true_type { };
}
```

23.2.6.1 `vector` constructors, copy, and assignment [vector.cons]

```
vector(const Allocator& = Allocator());
```

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

2 *Complexity:*Constant.

```
requires DefaultConstructible<T> explicit vector(size_type n);
```

3 *Effects:*Constructs a `vector` with `n` default constructed elements.

4 ~~*Requires:* `T` shall be `DefaultConstructible`.~~

5 *Complexity:*Linear in `n`.

```
requires ConstructibleAsElement<Alloc, T, const T&>
explicit vector(size_type n, const T& value,
                const Allocator& = Allocator());
```

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

7 ~~*Requires:* `T` shall be `CopyConstructible`.~~

8 *Complexity:*Linear in `n`.

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, T, Iter::reference> && ConstructibleAsElement<Alloc, T, T&&>
  vector(InputIteratorIter first, InputIteratorIter last,
         const Allocator& = Allocator());
```

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

10 *Complexity:* Makes only N calls to the copy constructor of `T` (where N is the distance between `first` and `last`) and no reallocations if iterators first and last are of forward, bidirectional, or random access categories. It makes order `N` calls to the copy constructor of `T` and order $\log(N)$ reallocations if they are just input iterators.

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, T, Iter::reference>
  void assign(InputIteratorIter first, InputIteratorIter last);
```

11 *Effects:*

```
erase(begin(), end());
insert(begin(), first, last);
```

```
requires ConstructibleAsElement<Alloc, T, const T&>
  void assign(size_type n, const T& t);
```

12 *Effects:*

```
erase(begin(), end());
insert(begin(), n, t);
```

23.2.6.2 `vector` capacity [vector.capacity]

`size_type capacity() const;`

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

`requires ConstructibleAsElement<Alloc, T, T&&> void reserve(size_type n);`

2 *Requires:*If `value_type` has a move constructor, that constructor shall not throw any exceptions.

3 *Effects:* A directive that informs a `vector` of a planned change in size, so that it can manage the storage allocation accordingly. After `reserve()`, `capacity()` is greater or equal to the argument of `reserve` if reallocation happens; and equal to the previous value of `capacity()` otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of `reserve()`. If an exception is thrown, there are no effects.

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

5 *Throws:* `length_error` if *n* `> max_size()`.[3]

6 *Remarks:* Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during insertions that happen after a call to `reserve()` until the time when an insertion would make the size of the vector greater than the value of `capacity()`.

`void swap(vector<T,Alloc~~ator~~>&& x);`

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

8 *Complexity:* Constant time.

`requires DefaultConstructible<T> && ConstructibleAsElement<Alloc, T, T&&> void resize(size_type sz);`

9 *Effects:*If `sz < size()`, equivalent to `erase(begin() + sz, end());`. If `size() < sz`, appends `sz - size()` default constructed elements to the sequence.

10 *Requires:*`T` shall be `DefaultConstructible`.

`requires ConstructibleAsElement<Alloc, T, const T&> void resize(size_type sz, const T& c);`

11 *Effects:*

```
if (sz > size())
  insert(end(), sz-size(), c);
else if (sz < size())
  erase(begin()+sz, end());
else
  ;                    // do nothing
```

12 *Requires:*If `value_type` has a move constructor, that constructor shall not throw any exceptions.

[3] `reserve()` uses `Alloc@~~ator~~@::allocate()` which may throw an appropriate exception.

23.2.6.3 `vector` data [vector.data]

```
pointer         data();
const_pointer   data() const;
```

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

2 *Complexity:* Constant time.

3 *Throws:* Nothing.

23.2.6.4 `vector` modifiers [vector.modifiers]

```
requires ConstructibleAsElement<Alloc, T, const T&> && MoveAssignable<T>
  iterator insert(const_iterator position, const T& x);
requires ConstructibleAsElement<Alloc, T, T&&> && MoveAssignable<T>
  iterator insert(const_iterator position, T&& x);
requires ConstructibleAsElement<Alloc, T, const T&> && MoveAssignable<T>
  void insert(const_iterator position, size_type n, const T& x);
template <class InputIteratorInputIterator Iter>
    requires ConstructibleAsElement<Alloc, T, Iter::reference>
          && ConstructibleAsElement<Alloc, T, T&&> && MoveAssignable<T>
    void insert(const_iterator position,
                InputIteratorIter first, InputIteratorIter last);

template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...> && ConstructibleAsElement<Alloc, T, T&&>
  void push_back(Args&&... args);
template <class... Args>
  requires ConstructibleAsElement<Alloc, T, Args&&...> && ConstructibleAsElement<Alloc, T, T&&>
  iterator emplace(const_iterator position, Args&&... args);
```

1 *Requires:* If `value_type` has a move constructor, that constructor shall not throw any exceptions.

2 *Remarks:* Causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid. If an exception is thrown other than by the copy constructor or assignment operator of `T` or by any `InputIterator` operation there are no effects.

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

```
requires MoveAssignable<T> iterator erase(const_iterator position);
requires MoveAssignable<T> iterator erase(const_iterator first, const_iterator last);
```

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

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

6 *Throws:* Nothing unless an exception is thrown by the copy constructor or assignment operator of `T`.

23.2.6.5 `vector` specialized algorithms [vector.special]

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

1 *Effects:*

```
x.swap(y);
```

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

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

```
namespace std {
  template <classRandomAccessAllocator Allocator> class vector<bool, Allocator> {
  public:
    // types:
    typedef bool                                   const_reference;
    typedef implementation-defined                 iterator;       // See 23.1
    typedef implementation-defined                 const_iterator; // See 23.1
    typedef implementation-defined                 size_type;      // See 23.1
    typedef implementation-defined                 difference_type;// See 23.1
    typedef bool                                   value_type;
    typedef Allocator                              allocator_type;
    typedef implementation-defined                 pointer;
    typedef implementation-defined                 const_pointer;
    typedef std::reverse_iterator<iterator>       reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

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

    // construct/copy/destroy:
    explicit vector(const Allocator& = Allocator());
    explicit vector(size_type n, const bool& value = bool(),
                    const Allocator& = Allocator());
    template <class InputIteratorInputIterator Iter>
```

```
  requires Convertible<Iter::reference, bool>
  vector(InputIteratorIter first, InputIteratorIter last,
         const Allocator& = Allocator());
vector(const vector<bool,Allocator>& x);
vector(vector<bool,Allocator>&& x);
vector(const vector&, const Allocator&);
vector(vector&&, const Allocator&);
~vector();
vector<bool,Allocator>& operator=(const vector<bool,Allocator>& x);
vector<bool,Allocator>& operator=(vector<bool,Allocator>&& x);
template <class InputIteratorInputIterator Iter>
  requires Convertible<Iter::reference, bool>
  void assign(InputIteratorIter first, InputIteratorIter last);
void assign(size_type n, const bool& t);
allocator_type get_allocator() const;

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

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

// capacity:
size_type size() const;
size_type max_size() const;
void      resize(size_type sz, bool c = false);
size_type capacity() const;
bool      empty() const;
void      reserve(size_type n);

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

// modifiers:
```

```
    void push_back(const bool& x);
    void pop_back();
    iterator insert(const_iterator position, const bool& x);
    void     insert (const_iterator position, size_type n, const bool& x);
    template <class InputIteratorInputIterator Iter>
        requires Convertible<Iter::reference, bool>
        void insert(const_iterator position,
                    InputIteratorIter first, InputIteratorIter last);

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

}
```

2 Unless described below, all operations have the same requirements and semantics as the primary `vector` template, except that operations dealing with the `bool` value type map to bit values in the container storage.

3 There is no requirement that the data be stored as a contiguous allocation of `bool` values. A space-optimized representation of bits is recommended instead.

4 `reference` is a class that simulates the behavior of references of a single bit in `vector<bool>`. The conversion operator returns `true` when the bit is set, and `false` otherwise. The assignment operator sets the bit when the argument is (convertible to) `true` and clears it otherwise. `flip` reverses the state of the bit.

```
void flip();
```

5 *Effects:*Replaces each element in the container with its complement. It is unspecified whether the function has any effect on allocated but unused bits.

23.3 Associative containers [associative]

1 Headers `<map>` and `<set>`:

Header `<map>` synopsis

```
namespace std {
  template <classObjectType Key, classObjectType T,
            classPredicate<auto, Key, Key> Compare = less<Key>,
            classAllocator Allocator = allocator<pair<const Key, T> > >
    requires Destructible<Key> && Destructible<T> && CopyConstructible<Compare>
             && ConstructibleAsElement<Alloc, Compare, const Compare&>
             && ConstructibleAsElement<Alloc, Compare, Compare&&>
    class map;
  template <classEqualityComparable Key, classEqualityComparable T, class Compare, class Allocator>
    bool operator==(const map<Key,T,Compare,Allocator>& x,
                    const map<Key,T,Compare,Allocator>& y);
```

```
template <classLessThanComparable Key, classLessThanComparable T, class Compare, class Allocator>
  bool operator< (const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);
template <classEqualityComparable Key, classEqualityComparable T, class Compare, class Allocator>
  bool operator!=(const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);
template <classLessThanComparable Key, classLessThanComparable T, class Compare, class Allocator>
  bool operator> (const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);
template <classLessThanComparable Key, classLessThanComparable T, class Compare, class Allocator>
  bool operator>=(const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);
template <classLessThanComparable Key, classLessThanComparable T, class Compare, class Allocator>
  bool operator<=(const map<Key,T,Compare,Allocator>& x,
                  const map<Key,T,Compare,Allocator>& y);
template <classObjectType Key, classObjectType T, class Compare, class Allocator>
  void swap(map<Key,T,Compare,Allocator>& x,
            map<Key,T,Compare,Allocator>& y);
template <classObjectType Key, classObjectType T, class Compare, class Allocator>
  void swap(map<Key,T,Compare,Allocator&& x,
            map<Key,T,Compare,Allocator>& y);
template <classObjectType Key, classObjectType T, class Compare, class Allocator>
  void swap(map<Key,T,Compare,Allocator& x,
            map<Key,T,Compare,Allocator>&& y);

template <classObjectType Key, classObjectType T,
          classPredicate<auto, Key, Key> Compare = less<Key>,
          classAllocator Allocator = allocator<pair<const Key, T> > >
  requires Destructible<Key> && Destructible<T> && CopyConstructible<Compare>
           && ConstructibleAsElement<Alloc, Compare, const Compare&>
           && ConstructibleAsElement<Alloc, Compare, Compare&&>
  class multimap;
template <classEqualityComparable Key, classEqualityComparable T, class Compare, class Allocator>
  bool operator==(const multimap<Key,T,Compare,Allocator>& x,
                  const multimap<Key,T,Compare,Allocator>& y);
template <classLessThanComparable Key, classLessThanComparable T, class Compare, class Allocator>
  bool operator< (const multimap<Key,T,Compare,Allocator>& x,
                  const multimap<Key,T,Compare,Allocator>& y);
template <classEqualityComparable Key, classEqualityComparable T, class Compare, class Allocator>
  bool operator!=(const multimap<Key,T,Compare,Allocator>& x,
                  const multimap<Key,T,Compare,Allocator>& y);
template <classLessThanComparable Key, classLessThanComparable T, class Compare, class Allocator>
  bool operator> (const multimap<Key,T,Compare,Allocator>& x,
                  const multimap<Key,T,Compare,Allocator>& y);
template <classLessThanComparable Key, classLessThanComparable T, class Compare, class Allocator>
  bool operator>=(const multimap<Key,T,Compare,Allocator>& x,
                  const multimap<Key,T,Compare,Allocator>& y);
template <classLessThanComparable Key, classLessThanComparable T, class Compare, class Allocator>
  bool operator<=(const multimap<Key,T,Compare,Allocator>& x,
                  const multimap<Key,T,Compare,Allocator>& y);
```

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

Header `<set>` **synopsis**

```
  namespace std {
    template <classObjectType Key, classPredicate<auto, Key, Key> Compare = less<Key>,
              classAllocator Allocator = allocator<Key> >
      requires Destructible<Key> && CopyConstructible<Compare>
               && ConstructibleAsElement<Alloc, Compare, const Compare&>
               && ConstructibleAsElement<Alloc, Compare, Compare&&>
      class set;
    template <classEqualityComparable Key, class Compare, class Allocator>
      bool operator==(const set<Key,Compare,Allocator>& x,
                      const set<Key,Compare,Allocator>& y);
    template <classLessThanComparable Key, class Compare, class Allocator>
      bool operator< (const set<Key,Compare,Allocator>& x,
                      const set<Key,Compare,Allocator>& y);
    template <classEqualityComparable Key, class Compare, class Allocator>
      bool operator!=(const set<Key,Compare,Allocator>& x,
                      const set<Key,Compare,Allocator>& y);
    template <classLessThanComparable Key, class Compare, class Allocator>
      bool operator> (const set<Key,Compare,Allocator>& x,
                      const set<Key,Compare,Allocator>& y);
    template <classLessThanComparable Key, class Compare, class Allocator>
      bool operator>=(const set<Key,Compare,Allocator>& x,
                      const set<Key,Compare,Allocator>& y);
    template <classLessThanComparable Key, class Compare, class Allocator>
      bool operator<=(const set<Key,Compare,Allocator>& x,
                      const set<Key,Compare,Allocator>& y);
    template <classObjectType Key, class Compare, class Allocator>
      void swap(set<Key,Compare,Allocator>& x,
                set<Key,Compare,Allocator>& y);
    template <classObjectType Key, class T, class Compare, class Allocator>
      void swap(set<Key,T,Compare,Allocator&& x,
                set<Key,T,Compare,Allocator>& y);
    template <classObjectType Key, class T, class Compare, class Allocator>
      void swap(set<Key,T,Compare,Allocator& x,
                set<Key,T,Compare,Allocator>&& y);

    template <classObjectType Key, classPredicate<auto, Key, Key> Compare = less<Key>,
              classAllocator Allocator = allocator<Key> >
```

```
    requires Destructible<Key> && CopyConstructible<Compare>
             && ConstructibleAsElement<Alloc, Compare, const Compare&>
             && ConstructibleAsElement<Alloc, Compare, Compare&&>
    class multiset;
  template <classEqualityComparable Key, class Compare, class Allocator>
    bool operator==(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <classLessThanComparable Key, class Compare, class Allocator>
    bool operator< (const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <classEqualityComparable Key, class Compare, class Allocator>
    bool operator!=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <classLessThanComparable Key, class Compare, class Allocator>
    bool operator> (const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <classLessThanComparable Key, class Compare, class Allocator>
    bool operator>=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <classLessThanComparable Key, class Compare, class Allocator>
    bool operator<=(const multiset<Key,Compare,Allocator>& x,
                    const multiset<Key,Compare,Allocator>& y);
  template <classObjectType Key, class Compare, class Allocator>
    void swap(multiset<Key,Compare,Allocator>& x,
              multiset<Key,Compare,Allocator>& y);
  template <classObjectType Key, class T, class Compare, class Allocator>
    void swap(multiset<Key,T,Compare,Allocator&& x,
              multiset<Key,T,Compare,Allocator>& y);
  template <classObjectType Key, class T, class Compare, class Allocator>
    void swap(multiset<Key,T,Compare,Allocator& x,
              multiset<Key,T,Compare,Allocator>&& y);
}
```

23.3.1 Class template `map` [map]

1 A `map` is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type `T` based on the keys. The `map` class supports bidirectional iterators.

2 A `map` satisfies all of the requirements of a container, of a reversible container (23.1), of an associative container ([associative.reqmts]), and of an allocator-aware container (Table [tab:containers.allocatoraware]). A `map` also provides most operations described in ([associative.reqmts]) for unique keys. This means that a `map` supports the `a_uniq` operations in ([associative.reqmts]) but not the `a_eq` operations. For a `map<Key,T>` the `key_type` is `Key` and the `value_type` is `pair<const Key,T>`. Descriptions are provided here only for operations on `map` that are not described in one of those tables or for operations where there is additional semantic information.

```
namespace std {
  template <classObjectType Key, classObjectType T,
            classPredicate<auto, Key, Key> Compare = less<Key>,
            classAllocator Allocator = allocator<pair<const Key, T> > >
  requires Destructible<Key> && Destructible<T> && CopyConstructible<Compare>
```

```
        && ConstructibleAsElement<Alloc, Compare, const Compare&>
        && ConstructibleAsElement<Alloc, Compare, Compare&&>
class map {
public:
  // types:
  typedef Key                                    key_type;
  typedef T                                      mapped_type;
  typedef pair<const Key, T>                     value_type;
  typedef Compare                                key_compare;
  typedef Allocator                              allocator_type;
  typedef typename Allocator::reference          reference;
  typedef typename Allocator::const_reference    const_reference;
  typedef implementation-defined                 iterator;       // See 23.1
  typedef implementation-defined                 const_iterator; // See 23.1
  typedef implementation-defined                 size_type;      // See 23.1
  typedef implementation-defined                 difference_type;// See 23.1
  typedef typename Allocator::pointer            pointer;
  typedef typename Allocator::const_pointer      const_pointer;
  typedef std::reverse_iterator<iterator>        reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

  class value_compare
    : public binary_function<value_type,value_type,bool> {
  friend class map;
  protected:
    Compare comp;
    value_compare(Compare c) : comp(c) {}
  public:
    bool operator()(const value_type& x, const value_type& y) const {
      return comp(x.first, y.first);
    }
  };

  // 23.3.1.1 construct/copy/destroy:
  explicit map(const Compare& comp = Compare(),
               const Allocator& = Allocator());
  template <class InputIteratorInputIterator Iter>
    requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
    map(InputIteratorIter first, InputIteratorIter last,
        const Compare& comp = Compare(), const Allocator& = Allocator());
  requires ConstructibleAsElement<Alloc, value_type, const value_type&>
    map(const map<Key,T,Compare,Allocator>& x);
  map(map<Key,T,Compare,Allocator>&& x);
  map(const Allocator&);
  requires ConstructibleAsElement<Alloc, value_type, const value_type&>
    map(const map&, const Allocator&);
  map(map&&, const Allocator&);
 ~map();
  requires ConstructibleAsElement<Alloc, value_type, const value_type&>
    map<Key,T,Compare,Allocator>& operator=(const map<Key,T,Compare,Allocator>& x);
```

```
map<Key,T,Compare,Allocator>&
  operator=(map<Key,T,Compare,Allocator>&& x);
allocator_type get_allocator() const;

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

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

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

// capacity:
bool      empty() const;
size_type size() const;
size_type max_size() const;

// 23.3.1.2 element access:
requires DefaultConstructible<T> && CopyConstructible<Key> T& operator[](const key_type& x);
requires DefaultConstructible<T> && MoveConstructible<Key> T& operator[](key_type&& x);
T&       at(const key_type& x);
const T& at(const key_type& x) const;

// modifiers:
template <class... Args>
  requires ConstructibleAsElement<Alloc, value_type, Args&&...>
  pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
  requires ConstructibleAsElement<Alloc, value_type, Args&&...>
  iterator emplace(const_iterator position, Args&&... args);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  pair<iterator, bool> insert(const value_type& x);
template <class P>
  requires ConstructibleAsElement<Alloc, value_type, P&&> && Convertible<P, value_type>
  pair<iterator, bool> insert(P&& x);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  iterator insert(const_iterator position, const value_type& x);
template <class P>
  requires ConstructibleAsElement<Alloc, value_type, P&&> && Convertible<P, value_type>
  iterator insert(const_iterator position, P&&);
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
```

```
    void insert(InputIteratorIter first, InputIteratorIter last);

  iterator  erase(const_iterator position);
  size_type erase(const key_type& x);
  iterator  erase(const_iterator first, const_iterator last);
  void swap(map<Key,T,Compare,Allocator>&&);
  void clear();

  // observers:
  key_compare   key_comp() const;
  value_compare value_comp() const;

  // 23.3.1.4 map operations:
  iterator       find(const key_type& x);
  const_iterator find(const key_type& x) const;
  size_type      count(const key_type& x) const;

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

  pair<iterator,iterator>
    equal_range(const key_type& x);
  pair<const_iterator,const_iterator>
    equal_range(const key_type& x) const;
};

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

// specialized algorithms:
template <classObjectType Key, classObjectType T, class Compare, class Allocator>
  void swap(map<Key,T,Compare,Allocator>& x,
```

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

    template <class Key, class T, class Compare, class Alloc>
      struct constructible_with_allocator_suffix<
        map<Key, T, Compare, Alloc> >
          : true_type { };
  }
```

23.3.1.1 `map` constructors, copy, and assignment [map.cons]

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

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

2 *Complexity:* Constant.

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  map(InputIteratorIter first, InputIteratorIter last,
      const Compare& comp = Compare(), const Allocator& = Allocator());
```

3 ~~*Requires:* If the iterator's dereference operator returns an lvalue or a const rvalue `pair<key_type, mapped_type>`, then both `key_type` and `mapped_type` shall be `CopyConstructible`.~~

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

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

23.3.1.2 `map` element access [map.access]

```
requires DefaultConstructible<T> && CopyConstructible<Key> T& operator[](const key_type& x);
```

1 *Effects:* If there is no key equivalent to `x` in the map, inserts `value_type(x, T())` into the map.

2 ~~*Requires:* `key_type` shall be `CopyConstructible` and `mapped_type` shall be `DefaultConstructible`.~~

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

4 *Complexity:* logarithmic.

```
requires DefaultConstructible<T> && MoveConstructible<Key> T& operator[](key_type&& x);
```

5 *Effects:*If there is no key equivalent to `x` in the map, inserts `value_type(std::move(x), T())` into the map.

6 ~~*Requires:* `mapped_type` shall be `DefaultConstructible`.~~

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

8 *Complexity:*logarithmic.

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

9 *Returns:*A reference to the element whose key is equivalent to `x`.

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

11 *Complexity:*logarithmic.

23.3.1.3 `map` modifiers [map.modifiers]

```
template <class P>
  requires ConstructibleAsElement<Alloc, value_type, P&&> && Convertible<P, value_type>
  pair<iterator, bool> insert(P&& x);
template <class P>
  requires ConstructibleAsElement<Alloc, value_type, P&&> && Convertible<P, value_type>
  pair<iterator, bool> insert(const_iterator position, P&& x);
```

1 ~~*Requires:* `P` shall be convertible to `value_type`.~~

If `P` is instantiated as a reference type, then the argument `x` is copied from. Otherwise `x` is considered to be an rvalue as it is converted to `value_type` and inserted into the `map`. Specifically, in such cases `CopyConstructible` is not required of `key_type` or `mapped_type` unless the conversion from `P` specifically requires it (e.g. if `P` is a `tuple<const key_type, mapped_type>`, then `key_type` must be `CopyConstructible`). The signature taking `InputIterator` parameters does not require `CopyConstructible` of either `key_type` or `mapped_type` if the dereferenced `InputIterator` returns a non-const rvalue `pair<key_type,mapped_type>`. Otherwise `CopyConstructible` is required for both `key_type` and `mapped_type`.

23.3.1.4 `map` operations [map.ops]

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

iterator       lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

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

pair<iterator, iterator>
  equal_range(const key_type &x);
pair<const_iterator, const_iterator>
```

```
equal_range(const key_type& x) const;
```

1 The `find`, `lower_bound`, `upper_bound` and `equal_range` member functions each have two versions, one const and the other non-const. In each case the behavior of the two functions is identical except that the const version returns a `const_iterator` and the non-const version an `iterator` ([associative.reqmts]).

23.3.1.5 `map` specialized algorithms [map.special]

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

1 *Effects:*

```
x.swap(y);
```

23.3.2 Class template `multimap` [multimap]

1 A `multimap` is an associative container that supports equivalent keys (possibly containing multiple copies of the same key value) and provides for fast retrieval of values of another type `T` based on the keys. The `multimap` class supports bidirectional iterators.

2 A `multimap` satisfies all of the requirements of a container and of a reversible container (23.1), of an associative container ([associative.reqmts]), and of an allocator-aware container (Table [tab:containers.allocatoraware]). A `multimap` also provides most operations described in ([associative.reqmts]) for equal keys. This means that a `multimap` supports the `a_eq` operations in ([associative.reqmts]) but not the `a_uniq` operations. For a `multimap<Key,T>` the `key_type` is `Key` and the `value_type` is `pair<const Key,T>`. Descriptions are provided here only for operations on `multimap` that are not described in one of those tables or for operations where there is additional semantic information.

```
namespace std {
  template <classObjectType Key, classObjectType T,
            classPredicate<auto, Key, Key> Compare = less<Key>,
            classAllocator Allocator = allocator<pair<const Key, T> > >
    requires Destructible<Key> && Destructible<T> && CopyConstructible<Compare>
          && ConstructibleAsElement<Alloc, Compare, const Compare&>
          && ConstructibleAsElement<Alloc, Compare, Compare&&>
  class multimap {
  public:
    // types:
    typedef Key                                  key_type;
    typedef T                                    mapped_type;
    typedef pair<const Key,T>                    value_type;
    typedef Compare                              key_compare;
```

```
typedef Allocator                              allocator_type;
typedef typename Allocator::reference          reference;
typedef typename Allocator::const_reference    const_reference;
typedef implementation-defined                 iterator;        // See 23.1
typedef implementation-defined                 const_iterator;  // See 23.1
typedef implementation-defined                 size_type;       // See 23.1
typedef implementation-defined                 difference_type;// See 23.1
typedef typename Allocator::pointer            pointer;
typedef typename Allocator::const_pointer      const_pointer;
typedef std::reverse_iterator<iterator>        reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

class value_compare
  : public binary_function<value_type,value_type,bool> {
friend class multimap;
protected:
  Compare comp;
  value_compare(Compare c) : comp(c) { }
public:
  bool operator()(const value_type& x, const value_type& y) const {
    return comp(x.first, y.first);
  }
};

// construct/copy/destroy:
explicit multimap(const Compare& comp = Compare(),
                  const Allocator& = Allocator());
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  multimap(InputIteratorIter first, InputIteratorIter last,
      const Compare& comp = Compare(), const Allocator& = Allocator());
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  multimap(const multimap<Key,T,Compare,Allocator>& x);
multimap(multimap<Key,T,Compare,Allocator>&& x);
multimap(const Allocator&);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  multimap(const multimap&, const Allocator&);
multimap(multimap&&, const Allocator&);
~multimap();
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  multimap<Key,T,Compare,Allocator>& operator=(const multimap<Key,T,Compare,Allocator>& x);
multimap<Key,T,Compare,Allocator>&
  operator=(const multimap<Key,T,Compare,Allocator>&& x);
allocator_type get_allocator() const;

// iterators:
iterator              begin();
const_iterator        begin() const;
iterator              end();
const_iterator        end() const;
```

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

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

// capacity:
bool           empty() const;
size_type      size() const;
size_type      max_size() const;

// modifiers:
template <class... Args>
  requires ConstructibleAsElement<Alloc, value_type, Args&&...>
  iterator emplace(Args&&... args);
template <class... Args>
  requires ConstructibleAsElement<Alloc, value_type, Args&&...>
  iterator emplace(const_iterator position, Args&&... args);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  iterator insert(const value_type& x);
template <class P>
  requires ConstructibleAsElement<Alloc, value_type, P&&> && Convertible<P, value_type>
  iterator insert(P&& x);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  iterator insert(const_iterator position, const value_type& x);
template <class P>
  requires ConstructibleAsElement<Alloc, value_type, P&&> && Convertible<P, value_type>
  iterator insert(const_iterator position, P&& x);
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  void insert(InputIteratorIter first, InputIteratorIter last);

iterator  erase(const_iterator position);
size_type erase(const key_type& x);
iterator  erase(const_iterator first, const_iterator last);
void swap(multimap<Key,T,Compare,Allocator>&&);
void clear();

// observers:
key_compare    key_comp() const;
value_compare  value_comp() const;

// map operations:
iterator       find(const key_type& x);
const_iterator find(const key_type& x) const;
```

```
  size_type      count(const key_type& x) const;

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

  pair<iterator,iterator>
    equal_range(const key_type& x);
  pair<const_iterator,const_iterator>
    equal_range(const key_type& x) const;
};

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

// specialized algorithms:
template <classObjectType Key, classObjectType T, class Compare, class Allocator>
  void swap(multimap<Key,T,Compare,Allocator>& x,
            multimap<Key,T,Compare,Allocator>& y);
template <classObjectType Key, classObjectType T, class Compare, class Allocator>
  void swap(multimap<Key,T,Compare,Allocator&& x,
            multimap<Key,T,Compare,Allocator>& y);
template <classObjectType Key, classObjectType T, class Compare, class Allocator>
  void swap(multimap<Key,T,Compare,Allocator& x,
            multimap<Key,T,Compare,Allocator>&& y);

template <class Key, class T, class Compare, class Alloc>
  struct constructible_with_allocator_suffix<
    multimap<Key, T, Compare, Alloc> >
      : true_type { };
}
```

23.3.2.1 `multimap` constructors [multimap.cons]

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

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

2 *Complexity:* Constant.

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  multimap(InputIteratorIter first, InputIteratorIter last,
      const Compare& comp = Compare(), const Allocator& = Allocator());
```

3 ~~*Requires:* If the iterator's dereference operator returns an lvalue or a const rvalue `pair<key_type, mapped_type>`, then both `key_type` and `mapped_type` shall be `CopyConstructible`.~~

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

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

23.3.2.2 `multimap` modifiers [multimap.modifiers]

```
template <class P>
  requires ConstructibleAsElement<Alloc, value_type, P&&> && Convertible<P, value_type>
  iterator insert(P&& x);
template <class P>
  requires ConstructibleAsElement<Alloc, value_type, P&&> && Convertible<P, value_type>
  iterator insert(const_iterator position, P&& x);
```

1 ~~*Requires:* `P` shall be convertible to `value_type`.~~

If `P` is instantiated as a reference type, then the argument `x` is copied from. Otherwise `x` is considered to be an rvalue as it is converted to `value_type` and inserted into the `map`. Specifically, in such cases `CopyConstructible` is not required of `key_type` or tcodemapped_type unless the conversion from `P` specifically requires it (e.g. if `P` is a `tuple<const key_type, mapped_type>`, then `key_type` must be `CopyConstructible`). The signature taking `InputIterator` parameters does not require `CopyConstructible` of either `key_type` or `mapped_type` if the dereferenced `InputIterator` returns a non-const rvalue `pair<key_type,mapped_type>`. Otherwise `CopyConstructible` is required for both `key_type` and `mapped_type`.

23.3.2.3 `multimap` operations [multimap.ops]

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

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

```
pair<iterator, iterator>
  equal_range(const key_type& x);
pair<const_iterator, const_iterator>
  equal_range(const key_type& x) const;
```

1 The `find`, `lower_bound`, `upper_bound`, and `equal_range` member functions each have two versions, one const and one non-const. In each case the behavior of the two versions is identical except that the const version returns a `const_iterator` and the non-const version an `iterator` ([associative.reqmts]).

23.3.2.4 `multimap` specialized algorithms [multimap.special]

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

1 *Effects:*

```
x.swap(y);
```

23.3.3 Class template `set` [set]

1 A `set` is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves. Class `set` supports bidirectional iterators.

2 A `set` satisfies all of the requirements of a container, of a reversible container (23.1), of an associative container ([associative.reqmts]), and of an allocator-aware container (Table [tab:containers.allocatoraware]). A `set` also provides most operations described in ([associative.reqmts]) for unique keys. This means that a `set` supports the `a_uniq` operations in ([associative.reqmts]) but not the `a_eq` operations. For a `set<Key>` both the `key_type` and `value_type` are `Key`. Descriptions are provided here only for operations on `set` that are not described in one of these tables and for operations where there is additional semantic information.

```
namespace std {
  template <classObjectType Key, classPredicate<auto, Key, Key> Compare = less<Key>,
            classAllocator Allocator = allocator<Key> >
  requires Destructible<Key> && CopyConstructible<Compare>
           && ConstructibleAsElement<Alloc, Compare, const Compare&>
           && ConstructibleAsElement<Alloc, Compare, Compare&&>
  class set {
  public:
    // types:
    typedef Key                                   key_type;
    typedef Key                                   value_type;
    typedef Compare                               key_compare;
```

```
typedef Compare                                   value_compare;
typedef Allocator                                 allocator_type;
typedef typename Allocator::reference             reference;
typedef typename Allocator::const_reference       const_reference;
typedef implementation-defined                    iterator;        // See 23.1
typedef implementation-defined                    const_iterator;  // See 23.1
typedef implementation-defined                    size_type;       // See 23.1
typedef implementation-defined                    difference_type; // See 23.1
typedef typename Allocator::pointer               pointer;
typedef typename Allocator::const_pointer         const_pointer;
typedef std::reverse_iterator<iterator>           reverse_iterator;
typedef std::reverse_iterator<const_iterator>     const_reverse_iterator;

// 23.3.3.1 construct/copy/destroy:
explicit set(const Compare& comp = Compare(),
             const Allocator& = Allocator());
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  set(InputIteratorIter first, InputIteratorIter last,
      const Compare& comp = Compare(), const Allocator& = Allocator());
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  set(const set<Key,Compare,Allocator>& x);
set(set<Key,Compare,Allocator>&& x);
set(const Allocator&);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  set(const set&, const Allocator&);
set(set&&, const Allocator&);
~set();

requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  set<Key,Compare,Allocator>& operator=(const set<Key,Compare,Allocator>& x);
set<Key,Compare,Allocator>& operator=(set<Key,Compare,Allocator>&& x);
allocator_type get_allocator() const;

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

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

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

```
  // capacity:
  bool          empty() const;
  size_type     size() const;
  size_type     max_size() const;

  // modifiers:
  template <class... Args>
    requires ConstructibleAsElement<Alloc, value_type, Args&&...>
    pair<iterator, bool> emplace(Args&&... args);
  template <class... Args>
    requires ConstructibleAsElement<Alloc, value_type, Args&&...>
    iterator emplace(const_iterator position, Args&&... args);
  requires ConstructibleAsElement<Alloc, value_type, const value_type&>
    pair<iterator,bool> insert(const value_type& x);
  requires ConstructibleAsElement<Alloc, value_type, value_type&&>
    pair<iterator,bool> insert(value_type&& x);
  requires ConstructibleAsElement<Alloc, value_type, const value_type&>
    iterator insert(const_iterator position, const value_type& x);
  requires ConstructibleAsElement<Alloc, value_type, value_type&&>
    iterator insert(const_iterator position, value_type&& x);
  template <class InputIteratorInputIterator Iter>
    requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
    void insert(InputIteratorIter first, InputIteratorIter last);

  iterator  erase(const_iterator position);
  size_type erase(const key_type& x);
  iterator  erase(const_iterator first, const_iterator last);
  void swap(set<Key,Compare,Allocator>&);
  void clear();

  // observers:
  key_compare   key_comp() const;
  value_compare value_comp() const;

  // set operations:
  iterator        find(const key_type& x);
  const_iterator  find(const key_type& x) const;

  size_type count(const key_type& x) const;

  iterator        lower_bound(const key_type& x);
  const_iterator  lower_bound(const key_type& x) const;

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

  pair<iterator,iterator>             equal_range(const key_type& x);
  pair<const_iterator,const_iterator> equal_range(const key_type& x) const;
};
```

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

  // specialized algorithms:
  template <classObjectType Key, class Compare, class Allocator>
    void swap(set<Key,Compare,Allocator>& x,
              set<Key,Compare,Allocator>& y);
  template <classObjectType Key, class Compare, class Allocator>
    void swap(set<Key,Compare,Allocator&& x,
              set<Key,Compare,Allocator>& y);
  template <classObjectType Key, class Compare, class Allocator>
    void swap(set<Key,Compare,Allocator& x,
              set<Key,Compare,Allocator>&& y);

  template <class Key, class Compare, class Alloc>
    struct constructible_with_allocator_suffix<
      set<Key, Compare, Alloc> >
        : true_type { };
}
```

23.3.3.1 `set` constructors, copy, and assignment [set.cons]

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

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

2 *Complexity:* Constant.

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  set(InputIteratorIter first, InputIteratorIter last,
      const Compare& comp = Compare(), const Allocator& = Allocator());
```

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

the range [*first* , *last*).

4 @~~*Requires:* If the iterator's dereference operator returns an lvalue or a non-const rvalue, then Key shall be CopyConstructible.~~

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

23.3.3.2 set specialized algorithms [set.special]

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

1 *Effects:*

```
x.swap(y);
```

23.3.4 Class template multiset [multiset]

1 A `multiset` is an associative container that supports equivalent keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves. Class `multiset` supports bidirectional iterators.

2 A `multiset` satisfies all of the requirements of a container, of a reversible container (23.1), of an associative container ([associative.reqmts]), and of an allocator-aware container (Table [tab:containers.allocatoraware]). `multiset` also provides most operations described in ([associative.reqmts]) for duplicate keys. This means that a `multiset` supports the `a_eq` operations in ([associative.reqmts]) but not the `a_uniq` operations. For a `multiset<Key>` both the `key_type` and `value_type` are `Key`. Descriptions are provided here only for operations on `multiset` that are not described in one of these tables and for operations where there is additional semantic information.

```
namespace std {
  template <classObjectType Key, classPredicate<auto, Key, Key> Compare = less<Key>,
            classAllocator Allocator = allocator<Key> >
  requires Destructible<Key> && CopyConstructible<Compare>
           && ConstructibleAsElement<Alloc, Compare, const Compare&>
           && ConstructibleAsElement<Alloc, Compare, Compare&&>
  class multiset {
  public:
    // types:
    typedef Key                                   key_type;
    typedef Key                                   value_type;
    typedef Compare                               key_compare;
    typedef Compare                               value_compare;
```

```
typedef Allocator                                allocator_type;
typedef typename Allocator::reference            reference;
typedef typename Allocator::const_reference      const_reference;
typedef implementation-defined                   iterator;        // See 23.1
typedef implementation-defined                   const_iterator;  // See 23.1
typedef implementation-defined                   size_type;       // See 23.1
typedef implementation-defined                   difference_type; // See 23.1
typedef typename Allocator::pointer              pointer;
typedef typename Allocator::const_pointer        const_pointer;
typedef std::reverse_iterator<iterator>          reverse_iterator;
typedef std::reverse_iterator<const_iterator>    const_reverse_iterator;

// construct/copy/destroy:
explicit multiset(const Compare& comp = Compare(),
                  const Allocator& = Allocator());
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  multiset(InputIteratorIter first, InputIteratorIter last,
           const Compare& comp = Compare(),
           const Allocator& = Allocator());
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  multiset(const multiset<Key,Compare,Allocator>& x);
multiset(multiset<Key,Compare,Allocator>&& x);
multiset(const Allocator&);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  multiset(const multiset&, const Allocator&);
multiset(multiset&&, const Allocator&);
~multiset();
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  multiset<Key,Compare,Allocator>& operator=(const multiset<Key,Compare,Allocator>& x);
multiset<Key,Compare,Allocator>& operator=(multiset<Key,Compare,Allocator>&& x);
allocator_type get_allocator() const;

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

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

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

// capacity:
```

```
  bool           empty() const;
  size_type      size() const;
  size_type      max_size() const;

  // modifiers:
  template <class... Args>
    requires ConstructibleAsElement<Alloc, value_type, Args&&...>
    iterator emplace(Args&&... args);
  template <class... Args>
    requires ConstructibleAsElement<Alloc, value_type, Args&&...>
    iterator emplace(const_iterator position, Args&&... args);
  requires ConstructibleAsElement<Alloc, value_type, const value_type&>
    iterator insert(const value_type& x);
  requires ConstructibleAsElement<Alloc, value_type, value_type&&>
    iterator insert(value_type&& x);
  requires ConstructibleAsElement<Alloc, value_type, const value_type&>
    iterator insert(const_iterator position, const value_type& x);
  requires ConstructibleAsElement<Alloc, value_type, value_type&&>
    iterator insert(const_iterator position, value_type&& x);
  template <class InputIteratorInputIterator Iter>
    requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
    void insert(InputIteratorIter first, InputIteratorIter last);

  iterator  erase(const_iterator position);
  size_type erase(const key_type& x);
  iterator  erase(const_iterator first, const_iterator last);
  void swap(multiset<Key,Compare,Allocator>&&);
  void clear();

  // observers:
  key_compare   key_comp() const;
  value_compare value_comp() const;

  // set operations:
  iterator        find(const key_type& x);
  const_iterator  find(const key_type& x) const;

  size_type count(const key_type& x) const;

  iterator        lower_bound(const key_type& x);
  const_iterator  lower_bound(const key_type& x) const;

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

  pair<iterator,iterator>             equal_range(const key_type& x);
  pair<const_iterator,const_iterator> equal_range(const key_type& x) const;
};

template <classEqualityComparable Key, class Compare, class Allocator>
```

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

  // specialized algorithms:
  template <classObjectType Key, class Compare, class Allocator>
    void swap(multiset<Key,Compare,Allocator>& x,
              multiset<Key,Compare,Allocator>& y);
  template <classObjectType Key, class Compare, class Allocator>
    void swap(multiset<Key,Compare,Allocator&& x,
              multiset<Key,Compare,Allocator>& y);
  template <classObjectType Key, class Compare, class Allocator>
    void swap(multiset<Key,Compare,Allocator& x,
              multiset<Key,Compare,Allocator>&& y);

  template <class Key, class Compare, class Alloc>
    struct constructible_with_allocator_suffix<
      multiset<Key, Compare, Alloc> >
        : true_type { };
}
```

23.3.4.1 `multiset` constructors [multiset.cons]

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

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

2 *Complexity:* Constant.

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  multiset(InputIteratorIter first, InputIteratorIter last,
           const Compare& comp = Compare(),
           const Allocator& = Allocator());
```

3 ~~*Requires:* If the iterator's dereference operator returns an lvalue or a const rvalue, then Key shall be CopyConstructible.~~

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

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

23.3.4.2 `multiset` specialized algorithms [multiset.special]

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

1 *Effects:*

```
x.swap(y);
```

23.4 Unordered associative containers [unord]

1 Headers `<unordered_map>` and `<unordered_set>`:

Header `<unordered_map>` synopsis

```
namespace std {
  // 23.4.1, class template unordered_map:
  template <classObjectType Key,
            classObjectType T,
            classCallable<auto, Key> Hash = hash<Key>,
            classPredicate<auto, Key, Key> Pred = std::equal_to<Key>,
            classRandomAccessAllocator Alloc = std::allocator<std::pair<const Key, T> > >
    requires SameType<Hash::result_type, std::size_t>
             && CopyConstructible<Hash> && CopyConstructible<Pred>
             && ConstructibleAsElement<Alloc, Pred, const Pred&>
             && ConstructibleAsElement<Alloc, Pred, Pred&&>
             && ConstructibleAsElement<Alloc, Hash, const Hash&>
             && ConstructibleAsElement<Alloc, Hash, Hash&&>
    class unordered_map;

  // 23.4.2, class template unordered_multimap:
  template <classObjectType Key,
            classObjectType T,
            classCallable<auto, Key> Hash = hash<Key>,
            classPredicate<auto, Key, Key> Pred = std::equal_to<Key>,
            classRandomAccessAllocator Alloc = std::allocator<std::pair<const Key, T> > >
    requires SameType<Hash::result_type, std::size_t>
```

```
              && CopyConstructible<Hash> && CopyConstructible<Pred>
              && ConstructibleAsElement<Alloc, Pred, const Pred&>
              && ConstructibleAsElement<Alloc, Pred, Pred&&>
              && ConstructibleAsElement<Alloc, Hash, const Hash&>
              && ConstructibleAsElement<Alloc, Hash, Hash&&>
    class unordered_multimap;

  template <classObjectType Key, classObjectType T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
              unordered_map<Key, T, Hash, Pred, Alloc>& y);

  template <classObjectType Key, classObjectType T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
} // namespace std
```

Header <unordered_set> synopsis

```
namespace std {
  // 23.4.3, class template unordered_set:
  template <classObjectType Value,
            classCallable<auto, Value> Hash = hash<Value>,
            classPredicate<auto, Value, Value> class Pred = std::equal_to<Value>,
            classRandomAccessAllocator Alloc = std::allocator<Value> >
    requires SameType<Hash::result_type, std::size_t>
              && CopyConstructible<Hash> && CopyConstructible<Pred>
              && ConstructibleAsElement<Alloc, Pred, const Pred&>
              && ConstructibleAsElement<Alloc, Pred, Pred&&>
              && ConstructibleAsElement<Alloc, Hash, const Hash&>
              && ConstructibleAsElement<Alloc, Hash, Hash&&>
    class unordered_set;

  // 23.4.4, class template unordered_multiset:
  template <classObjectType Value,
            classCallable<auto, Value> Hash = hash<Value>,
            classPredicate<auto, Value, Value> class Pred = std::equal_to<Value>,
            classRandomAccessAllocator Alloc = std::allocator<Value> >
    requires SameType<Hash::result_type, std::size_t>
              && CopyConstructible<Hash> && CopyConstructible<Pred>
              && ConstructibleAsElement<Alloc, Pred, const Pred&>
              && ConstructibleAsElement<Alloc, Pred, Pred&&>
              && ConstructibleAsElement<Alloc, Hash, const Hash&>
              && ConstructibleAsElement<Alloc, Hash, Hash&&>
    class unordered_multiset;

  template <classObjectType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
              unordered_set<Value, Hash, Pred, Alloc>& y);

  template <classObjectType Value, class Hash, class Pred, class Alloc>
```

```
    void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
              unordered_multiset<Value, Hash, Pred, Alloc>& y);
} // namespace std
```

23.4.1 Class template unordered_map [unord.map]

1 An unordered_map is an unordered associative container that supports unique keys (an unordered_map contains at most one of each key value) and that associates values of another type mapped_type with the keys.

2 An unordered_map satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table [tab:containers.allocatoraware]). It provides the operations described in the preceding requirements table for unique keys; that is, an unordered_map supports the a_uniq operations in that table, not the a_eq operations. For an unordered_map<Key, T> the key type is Key, the mapped type is T, and the value type is std::pair<const Key, T>.

3 This section only describes operations on unordered_map that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
  template <classObjectType Key,
            classObjectType T,
            classCallable<auto, Key> Hash = hash<Key>,
            classPredicate<auto, Key, Key> Pred = std::equal_to<Key>,
            classRandomAccessAllocator Alloc = std::allocator<std::pair<const Key, T> > >
  requires SameType<Hash::result_type, std::size_t>
           && CopyConstructible<Hash> && CopyConstructible<Pred>
           && ConstructibleAsElement<Alloc, Pred, const Pred&>
           && ConstructibleAsElement<Alloc, Pred, Pred&&>
           && ConstructibleAsElement<Alloc, Hash, const Hash&>
           && ConstructibleAsElement<Alloc, Hash, Hash&&>
  class unordered_map
  {
  public:
    // types
    typedef Key                                     key_type;
    typedef std::pair<const Key, T>                 value_type;
    typedef T                                       mapped_type;
    typedef Hash                                    hasher;
    typedef Pred                                    key_equal;
    typedef Alloc                                   allocator_type;
    typedef typename allocator_type::pointer        pointer;
    typedef typename allocator_type::const_pointer  const_pointer;
    typedef typename allocator_type::reference      reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef implementation-defined                  size_type;
    typedef implementation-defined                  difference_type;

    typedef implementation-defined                  iterator;
    typedef implementation-defined                  const_iterator;
    typedef implementation-defined                  local_iterator;
    typedef implementation-defined                  const_local_iterator;
```

```
// construct/destroy/copy
explicit unordered_map(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  unordered_map(InputIteratorIter f, InputIteratorIter l,
                size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  unordered_map(const unordered_map&);
unordered_map(const Allocator&);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  unordered_map(const unordered_map&, const Allocator&);
unordered_map(unordered_map&&, const Allocator&);
~unordered_map();
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  unordered_map& operator=(const unordered_map&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args>
  requires ConstructibleAsElement<Alloc, value_type, Args&&...>
  pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
  requires ConstructibleAsElement<Alloc, value_type, Args&&...>
  iterator emplace(const_iterator position, Args&&... args);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  std::pair<iterator, bool> insert(const value_type& obj);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  iterator insert(const_iterator hint, const value_type& obj);
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
```

```
    void insert(InputIteratorIter first, InputIteratorIter last);

  iterator erase(const_iterator position);
  size_type erase(const key_type& k);
  iterator erase(const_iterator first, const_iterator last);
  void clear();

  void swap(unordered_map&);

  // observers
  hasher hash_function() const;
  key_equal key_eq() const;

  // lookup
  iterator       find(const key_type& k);
  const_iterator find(const key_type& k) const;
  size_type count(const key_type& k) const;
  std::pair<iterator, iterator>             equal_range(const key_type& k);
  std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

  requires DefaultConstructible<T> && CopyConstructible<Key>
    mapped_type& operator[](const key_type& k);

  // bucket interface
  size_type bucket_count() const;
  size_type max_bucket_count() const;
  size_type bucket_size(size_type n);
  size_type bucket(const key_type& k) const;
  local_iterator begin(size_type n) const;
  const_local_iterator begin(size_type n) const;
  local_iterator end(size_type n);
  const_local_iterator end(size_type n) const;

  // hash policy
  float load_factor() const;
  float max_load_factor() const;
  void max_load_factor(float z);
  requires ConstructibleAsElement<Alloc, value_type, value_type&&> void rehash(size_type n);
};

template <classObjectType Key, classObjectType T, class Hash, class Pred, class Alloc>
  void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
            unordered_map<Key, T, Hash, Pred, Alloc>& y);

template <class Key, class T, class Hash, class Pred, class Alloc>
  struct constructible_with_allocator_suffix<
    unordered_map<Key, T, Hash, Pred, Compare, Alloc> >
      : true_type { };
}
```

23.4.1.1 `unordered_map` constructors [unord.map.cnstr]

```
explicit unordered_map(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty `unordered_map` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. `max_load_factor()` returns 1.0.

2 *Complexity:* Constant.

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  unordered_map(InputIteratorIter f, InputIteratorIter l,
                size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty `unordered_map` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range `[f, l)`. `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

23.4.1.2 `unordered_map` element access [unord.map.elem]

```
requires DefaultConstructible<T> && CopyConstructible<Key> mapped_type& operator[](const key_type& k);
```

1 *Effects:* If the `unordered_map` does not already contain an element whose key is equivalent to *k*, inserts the value `std::pair<const key_type, mapped_type>(k, mapped_type())`.

2 *Returns:* A reference to `x.second`, where `x` is the (unique) element whose key is equivalent to *k*.

23.4.1.3 `unordered_map` swap [unord.map.swap]

```
template <classObjectType Key, classObjectType T, class Hash, class Pred, class Alloc>
  void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
            unordered_map<Key, T, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y)`.

23.4.2 Class template `unordered_multimap` [unord.multimap]

1 An `unordered_multimap` is an unordered associative container that supports equivalent keys (an `unordered_multimap` may contain multiple copies of each key value) and that associates values of another type `mapped_type` with the keys.

2 An `unordered_multimap` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table [tab:containers.allocatoraware]). It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multimap` supports the `a_eq` operations in that table, not the `a_uniq` operations. For an `unordered_multimap<Key, T>` the `key type` is `Key`, the mapped type is `T`, and the value type is `std::pair<const Key, T>`.

3 This section only describes operations on `unordered_multimap` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
  template <classObjectType Key,
            classObjectType T,
            classCallable<auto, Key> Hash = hash<Key>,
            classPredicate<auto, Key, Key> Pred = std::equal_to<Key>,
            classRandomAccessAllocator Alloc = std::allocator<std::pair<const Key, T> > >
  requires SameType<Hash::result_type, std::size_t>
           && CopyConstructible<Hash> && CopyConstructible<Pred>
           && ConstructibleAsElement<Alloc, Pred, const Pred&>
           && ConstructibleAsElement<Alloc, Pred, Pred&&>
           && ConstructibleAsElement<Alloc, Hash, const Hash&>
           && ConstructibleAsElement<Alloc, Hash, Hash&&>
  class unordered_multimap
  {
  public:
    // types
    typedef Key                                        key_type;
    typedef std::pair<const Key, T>                    value_type;
    typedef T                                          mapped_type;
    typedef Hash                                       hasher;
    typedef Pred                                       key_equal;
    typedef Alloc                                      allocator_type;
    typedef typename allocator_type::pointer           pointer;
    typedef typename allocator_type::const_pointer     const_pointer;
    typedef typename allocator_type::reference         reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef implementation-defined                     size_type;
    typedef implementation-defined                     difference_type;

    typedef implementation-defined                     iterator;
    typedef implementation-defined                     const_iterator;
    typedef implementation-defined                     local_iterator;
    typedef implementation-defined                     const_local_iterator;

    // construct/destroy/copy
    explicit unordered_multimap(size_type n = implementation-defined,
                                const hasher& hf = hasher(),
                                const key_equal& eql = key_equal(),
                                const allocator_type& a = allocator_type());
    template <class InputIteratorInputIterator Iter>
      requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
```

```
  unordered_multimap(InputIteratorIter f, InputIteratorIter l,
                     size_type n = implementation-defined,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  unordered_multimap(const unordered_multimap&);
unordered_multimap(const Allocator&);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
   unordered_multimap(const unordered_multimap&, const Allocator&);
unordered_multimap(unordered_multimap&&, const Allocator&);
~unordered_multimap();
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  unordered_multimap& operator=(const unordered_multimap&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args>
  requires ConstructibleAsElement<Alloc, value_type, Args&&...>
  iterator emplace(Args&&... args);
template <class... Args>
  requires ConstructibleAsElement<Alloc, value_type, Args&&...>
  iterator emplace(const_iterator position, Args&&... args);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  iterator insert(const value_type& obj);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  iterator insert(const_iterator hint, const value_type& obj);
template <class InputIteratorInputIterator Iter>
  void insert(InputIteratorIter first, InputIteratorIter last);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_multimap&);
```

```
    // observers
    hasher hash_function() const;
    key_equal key_eq() const;

    // lookup
    iterator       find(const key_type& k);
    const_iterator find(const key_type& k) const;
    size_type count(const key_type& k) const;
    std::pair<iterator, iterator>             equal_range(const key_type& k);
    std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

    // bucket interface
    size_type bucket_count() const;
    size_type max_bucket_count() const;
    size_type bucket_size(size_type n);
    size_type bucket(const key_type& k) const;
    local_iterator begin(size_type n) const;
    const_local_iterator begin(size_type n) const;
    local_iterator end(size_type n);
    const_local_iterator end(size_type n) const;

    // hash policy
    float load_factor() const;
    float max_load_factor() const;
    void max_load_factor(float z);
    requires ConstructibleAsElement<Alloc, value_type, value_type&&> void rehash(size_type n);
  };

  template <classObjectType Key, classObjectType T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>& y);

  template <class Key, class T, class Hash, class Pred, class Alloc>
    struct constructible_with_allocator_suffix<
      unordered_multimap<Key, T, Hash, Pred, Alloc> >
        : true_type { };
}
```

23.4.2.1 `unordered_multimap` constructors [unord.multimap.cnstr]

```
explicit unordered_multimap(size_type n = implementation-defined,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty `unordered_multimap` using the specified hash function, key equality function, and allocator, and using at least n buckets. If n is not provided, the number of buckets is implementation defined. `max_load_factor()` returns 1.0.

2 *Complexity:* Constant.

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  unordered_multimap(InputIteratorIter f, InputIteratorIter l,
                     size_type n = implementation-defined,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty unordered_multimap using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range [*f*, *l*). max_load_factor() returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

23.4.2.2 unordered_multimap swap [unord.multimap.swap]

```
template <classObjectType Key, classObjectType T, class Hash, class Pred, class Alloc>
  void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
            unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
```

1 *Effects:* x.swap(y).

23.4.3 Class template unordered_set [unord.set]

1 An unordered_set is an unordered associative container that supports unique keys (an unordered_set contains at most one of each key value) and in which the elements' keys are the elements themselves.

2 An unordered_set satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table [tab:containers.allocatoraware]). It provides the operations described in the preceding requirements table for unique keys; that is, an unordered_set supports the a_uniq operations in that table, not the a_eq operations. For an unordered_set<Value> the key type and the value type are both Value. The iterator and const_iterator types are both const iterator types. It is unspecified whether they are the same type.

3 This section only describes operations on unordered_set that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
  template <classObjectType Value,
            classCallable<auto, Value> Hash = hash<Value>,
            classPredicate<auto, Value, Value> class Pred = std::equal_to<Value>,
            classRandomAccessAllocator Alloc = std::allocator<Value> >
  requires SameType<Hash::result_type, std::size_t>
           && CopyConstructible<Hash> && CopyConstructible<Pred>
           && ConstructibleAsElement<Alloc, Pred, const Pred&>
           && ConstructibleAsElement<Alloc, Pred, Pred&&>
           && ConstructibleAsElement<Alloc, Hash, const Hash&>
           && ConstructibleAsElement<Alloc, Hash, Hash&&>
  class unordered_set
  {
  public:
```

```
// types
typedef Value                                    key_type;
typedef Value                                    value_type;
typedef Hash                                     hasher;
typedef Pred                                     key_equal;
typedef Alloc                                    allocator_type;
typedef typename allocator_type::pointer         pointer;
typedef typename allocator_type::const_pointer   const_pointer;
typedef typename allocator_type::reference       reference;
typedef typename allocator_type::const_reference const_reference;
typedef implementation-defined                   size_type;
typedef implementation-defined                   difference_type;

typedef implementation-defined                   iterator;
typedef implementation-defined                   const_iterator;
typedef implementation-defined                   local_iterator;
typedef implementation-defined                   const_local_iterator;

// construct/destroy/copy
explicit unordered_set(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  unordered_set(InputIteratorIter f, InputIteratorIter l,
                size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  unordered_set(const unordered_set&);
unordered_set(const Allocator&);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  unordered_set(const unordered_set&, const Allocator&);
unordered_set(unordered_set&&, const Allocator&);
~unordered_set();
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  unordered_set& operator=(const unordered_set&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

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

```
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args>
  requires ConstructibleAsElement<Alloc, value_type, Args&&...>
  pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
  requires ConstructibleAsElement<Alloc, value_type, Args&&...>
  iterator emplace(const_iterator position, Args&&... args);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  std::pair<iterator, bool> insert(const value_type& obj);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  iterator insert(const_iterator hint, const value_type& obj);
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  void insert(InputIteratorIter first, InputIteratorIter last);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_set&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator       find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator>             equal_range(const key_type& k);
std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;

// hash policy
float load_factor() const;
float max_load_factor() const;
```

```
    void max_load_factor(float z);
    requires ConstructibleAsElement<Alloc, value_type, value_type&&> void rehash(size_type n);
  };

  template <classObjectType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
              unordered_set<Value, Hash, Pred, Alloc>& y);

  template <class Value, class Hash, class Pred, class Alloc>
    struct constructible_with_allocator_suffix<
      map<Value, Hash, Pred, Alloc> >
        : true_type { };
}
```

23.4.3.1 `unordered_set` constructors [unord.set.cnstr]

```
explicit unordered_set(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty `unordered_set` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. `max_load_factor()` returns 1.0.

2 *Complexity:* Constant.

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  unordered_set(InputIteratorIter f, InputIteratorIter l,
                size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty `unordered_set` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range `[`*f*`, `*l*`)`. `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

23.4.3.2 `unordered_set` swap [unord.set.swap]

```
template <classObjectType Value, class Hash, class Pred, class Alloc>
  void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
            unordered_set<Value, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y)`.

23.4.4 Class template `unordered_multiset` [unord.multiset]

1 An `unordered_multiset` is an unordered associative container that supports equivalent keys (an `unordered_multiset` may contain multiple copies of the same key value) and in which each element's key is the element itself.

2 An `unordered_multiset` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table [tab:containers.allocatoraware]). It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multiset` supports the `a_eq` operations in that table, not the `a_uniq` operations. For an `unordered_multiset<Value>` the `key type` and the value type are both `Value`. The `iterator` and `const_iterator` types are both const iterator types. It is unspecified whether they are the same type.

3 This section only describes operations on `unordered_multiset` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
  template <classObjectType Value,
            classCallable<auto, Value> Hash = hash<Value>,
            classPredicate<auto, Value, Value> class Pred = std::equal_to<Value>,
            classRandomAccessAllocator Alloc = std::allocator<Value> >
  requires SameType<Hash::result_type, std::size_t>
           && CopyConstructible<Hash> && CopyConstructible<Pred>
           && ConstructibleAsElement<Alloc, Pred, const Pred&>
           && ConstructibleAsElement<Alloc, Pred, Pred&&>
           && ConstructibleAsElement<Alloc, Hash, const Hash&>
           && ConstructibleAsElement<Alloc, Hash, Hash&&>
  class unordered_multiset
  {
  public:
    // types
    typedef Value                                    key_type;
    typedef Value                                    value_type;
    typedef Hash                                     hasher;
    typedef Pred                                     key_equal;
    typedef Alloc                                    allocator_type;
    typedef typename allocator_type::pointer         pointer;
    typedef typename allocator_type::const_pointer   const_pointer;
    typedef typename allocator_type::reference       reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef implementation-defined                   size_type;
    typedef implementation-defined                   difference_type;

    typedef implementation-defined                   iterator;
    typedef implementation-defined                   const_iterator;
    typedef implementation-defined                   local_iterator;
    typedef implementation-defined                   const_local_iterator;

    // construct/destroy/copy
    explicit unordered_multiset(size_type n = implementation-defined,
                                const hasher& hf = hasher(),
                                const key_equal& eql = key_equal(),
```

```
                            const allocator_type& a = allocator_type());
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  unordered_multiset(InputIteratorIter f, InputIteratorIter l,
                     size_type n = implementation-defined,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  unordered_multiset(const unordered_multiset&);
unordered_multiset(const Allocator&);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  unordered_multiset(const unordered_multiset&, const Allocator&);
unordered_multiset(unordered_multiset&&, const Allocator&);
~unordered_multiset();
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  unordered_multiset& operator=(const unordered_multiset&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args>
  requires ConstructibleAsElement<Alloc, value_type, Args&&...>
  iterator emplace(Args&&... args);
template <class... Args>
  requires ConstructibleAsElement<Alloc, value_type, Args&&...>
  iterator emplace(const_iterator position, Args&&... args);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  iterator insert(const value_type& obj);
requires ConstructibleAsElement<Alloc, value_type, const value_type&>
  iterator insert(const_iterator hint, const value_type& obj);
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::value_type>
  void insert(InputIteratorIter first, InputIteratorIter last);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
```

```
    void clear();

    void swap(unordered_multiset&);

    // observers
    hasher hash_function() const;
    key_equal key_eq() const;

    // lookup
    iterator       find(const key_type& k);
    const_iterator find(const key_type& k) const;
    size_type count(const key_type& k) const;
    std::pair<iterator, iterator>             equal_range(const key_type& k);
    std::pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

    // bucket interface
    size_type bucket_count() const;
    size_type max_bucket_count() const;
    size_type bucket_size(size_type n);
    size_type bucket(const key_type& k) const;
    local_iterator begin(size_type n) const;
    const_local_iterator begin(size_type n) const;
    local_iterator end(size_type n);
    const_local_iterator end(size_type n) const;

    // hash policy
    float load_factor() const;
    float max_load_factor() const;
    void max_load_factor(float z);
    requires ConstructibleAsElement<Alloc, value_type, value_type&&> void rehash(size_type n);
  };

  template <classObjectType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
              unordered_multiset<Value, Hash, Pred, Alloc>& y);

  template <class Value, class Hash, class Pred, class Alloc>
    struct constructible_with_allocator_suffix<
      unordered_multiset<Value, Hash, Pred, Alloc> >
        : true_type { };
}
```

23.4.4.1 `unordered_multiset` constructors [unord.multiset.cnstr]

```
explicit unordered_multiset(size_type n = implementation-defined,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty `unordered_multiset` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. `max_load_factor()` returns 1.0.

2 *Complexity:* Constant.

```
template <class InputIteratorInputIterator Iter>
  requires ConstructibleAsElement<Alloc, value_type, Iter::reference>
  unordered_multiset(InputIteratorIter f, InputIteratorIter l,
                     size_type n = implementation-defined,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty `unordered_multiset` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range `[f, l)`. `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

23.4.4.2 `unordered_multiset` swap [unord.multiset.swap]

```
template <classObjectType Value, class Hash, class Pred, class Alloc>
  void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
            unordered_multiset<Value, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y);`

Bibliography

[1] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek. Proposed wording for concepts (revision 5). Technical Report N2617=08-0127, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2008.