Concepts for the C++0x Standard Library: Containers

Douglas Gregor
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
dgregor@osl.iu.edu

Document number: N2085=06-0155
Date: 2006-09-09
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]. Most of the changes in this document have been verified to work with ConceptGCC and its modified Standard Library implementation. 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 working draft of the C++ standard (N2009). 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 24 Containers library [lib.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)
24.1 Requirements	`<container>`

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

?? Associative containers	`<map>`
	`<set>`

?? `bitset`	`<bitset>`

?? Unordered associative containers	`<unordered_map>`
	`<unordered_set>`

24.1 Container requirements [lib.container.requirements]

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

2 All of the complexity requirements in this clause are stated solely in terms of the number of operations on the contained objects. [*Example:* the copy constructor of type `vector <vector<int> >` has linear complexity, even though the complexity of copying each contained `vector<int>` is itself linear. — *end example*]

3 ~~The type of objects stored in these components shall meet the requirements of CopyConstructible types (20.1.3).~~

4 ~~Table 79 defines the Assignable requirement. Some containers require this property of the types to be stored in the container. T is the type used to instantiate the container, t is a value of T, and u is a value of (possibly const) T.~~

[[**Remove Table 79: Assignable requirements. Assignable is now a concept in Chapter 20.**]]

Header `<container>` synopsis

Note: Synchronize this with the rest of the text.

5 ~~In Tables 80 and 81, X denotes a container class containing objects of type T, a and b denote values of type X, u denotes an identifier and r denotes a value of X&.~~ The `Container` concept describes the requirements common to all containers.

```
concept Container<typename X> : DefaultConstructible<X>
{
  typename value_type =            X::value_type;
  typename reference =             X::reference;
  typename const_reference =       X::const_reference;
  InputIterator iterator =         X::iterator;
  InputIterator const_iterator =   X::const_iterator;
  SignedIntegral difference_type = X::difference_type;
  UnsignedIntegral size_type =     X::size_type;

  where Convertible<reference, value_type&> &&
        Convertible<const_reference, value_type const&>;

  where Convertible<iterator, const_iterator> &&
        SameType<iterator::value_type, value_type> &&
        SameType<const_iterator::value_type, value_type>;

  where SameType<difference_type, iterator::difference_type> &&
        SameType<difference_type, const_iterator::difference_type>;

  iterator       X::begin();
  const_iterator X::begin() const;
  iterator       X::end();
  const_iterator X::end() const;

  const_iterator X::cbegin() const;
  const_iterator X::cend() const;

  void X::swap(X&);

  size_type X::size() const;
  size_type X::max_size() const;
  bool X::empty() const;
}
```

[[**Remove Table 80: Container requirements**]]

In translating the requirements table into a concept, we have fixed numerous places where the requirements table required a non-constant container, but where a constant container would work. This should not break any existing code.

`X::X()`

6 *Postcondition:* `size() == 0`

7 *Complexity:* constant

```
X::~X()
```

8 *Remark:* the destructor is applied to every element of ~~a~~the container; all the memory is deallocated

9 *Complexity:* linear

```
iterator       X::begin();
const_iterator X::begin() const;
```

10 *Returns:* an iterator referring to the first element in the container. *Complexity:* constant

```
iterator       X::end();
const_iterator X::end() const;
```

12 *Returns:* returns an iterator which is the past-the-end value for the container. If the container is empty, then `begin() == end();`

13 *Complexity:* constant

```
const_iterator X::cbegin() const;
```

14 *Returns:* `const_cast<X const&>(*this).begin()`

15 *Complexity:* constant

```
const_iterator X::cend() const;
```

16 *Returns:* `const_cast<X const&>(*this).end()`

17 *Complexity:* constant

```
void X::swap(X& b)
```

18 *Effects:* `swap(*this, b)`

19 *Complexity:* should be constant

```
size_type X::size() const
```

20 *Returns:* returns the number of elements in the container.

21 *Complexity:* should be constant

22 *Remark:* ~~Its semantics~~The semantics of size() is defined by the rules of constructors, inserts, and erases.

```
size_type X::max_size() const
```

23 *Returns:* `size()` of the largest possible container

24 *Complexity:* should be constant

```
bool X::empty() const;
```

25 *Returns:* `size() == 0`

26 *Complexity:* constant

27 In the expressions

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

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

The requirements table for containers calls for copy construction, assignment, and various comparison operators. However, these operators only work when the value type supports copy construction, assignment, etc. To capture this behaviro, we state these requirements via concept map templates.

```
template<Container X>
where CopyConstructible<X::value_type>
concept_map CopyConstructible<X> { }
```

28 If the `value_type` of a container is `CopyConstructible`, the container shall be `CopyConstructible`

29 *Complexity:* linear

```
template<Container X>
where Assignable<X::value_type>
concept_map Assignable<X> { }
```

30 If the `value_type` of a container is `Assignable`, the container shall be `Assignable`

31 *Complexity:* linear

```
template<Container X>
where EqualityComparable<X::value_type>
concept_map EqualityComparable<X>
{
  bool operator==(X a, X b);
}
```

32 If the `value_type` of a container is `EqualityComparable`, the container shall be `EqualityComparable`

```
bool operator==(X a, X b);
```

33 *Effects:* == is an equivalence relation.

34 *Returns:* `a.size() == b.size() && equal(a.begin(), a.end(), b.begin()`

35 *Complexity:* linear

```
template<Container X>
where LessThanComparable<X::value_type>
concept_map LessThanComparable<X>
{
  bool operator<(X a, X b);
}
```

36 If the value_type of a container is LessThanComparable, the container shall be LessThanComparable

```
bool operator<(X a, X b);
```

37 *Returns:* `lexicographical_compare( a.begin(), a.end(), b.begin(), b.end())`

38 *Complexity:* linear

39 Copy constructors for all container types defined in this clause copy an allocator argument from their respective first parameters. All other constructors for these container types take an `Allocator&` argument (**??**), 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. In all container types defined in this clause, the member `get_allocator()` returns a copy of the Allocator object used to construct the container.[1)]

40 Containers that provide the ability to traverse their elements in reverse order are called reversible.

```
concept ReversibleContainer<typename X> : Container<X> {
  MutableBidirectionalIterator reverse_iterator       = X::reverse_iterator;
  BidirectionalIterator const_reverse_iterator = X::const_reverse_iterator;

  where MutableBidirectionalIterator<iterator> &&
        BidirectionalIterator<const_iterator>;

  where SameType<iterator::value_type, reverse_iterator::value_type> &&
        SameType<const_iterator::value_type, const_reverse_iterator::value_type>;

  reverse_iterator       X::rbegin();
  const_reverse_iterator X::rbegin() const;
  reverse_iterator       X::rend();
  const_reverse_iterator X::rend() const;
  const_reverse_iterator X::crbegin() const;
  const_reverse_iterator X::crend() const;
}
```

[[Remove Table 81: Reversible container requirements]]

41 If the iterator type of a container ~~belongs to the bidirectional or random access iterator categories~~is bidirectional or random access (**??**), ~~the container is called reversible and satisfies the additional requirements in Table 81~~the container is reversible.

```
template<Container X>
where MutableBidirectionalIterator<X::iterator> &&
      BidirectionalIterator<X::const_iterator>
concept_map ReversibleContainer<X>
{
  typedef std::reverse_iterator<X::iterator>       reverse_iterator;
  typedef std::reverse_iterator<X::const_iterator> const_reverse_iterator;
```

1) As specified in **??**, ~~paragraphs 4-5~~, the semantics described in this clause applies only to the case where allocators compare equal.

```
  reverse_iterator       X::rbegin();
  const_reverse_iterator X::rbegin() const;
  reverse_iterator       X::rend();
  const_reverse_iterator X::rend() const;
  const_reverse_iterator X::crbegin() const;
  const_reverse_iterator X::crend() const;
}
```

```
reverse_iterator       X::rbegin();
```

42 *Returns:* `reverse_iterator(end())`

```
const_reverse_iterator       X::rbegin() const;
```

43 *Returns:* `const_reverse_iterator(end())`

```
reverse_iterator       X::rend();
```

44 *Returns:* `const_reverse_iterator(begin())`

```
const_reverse_iterator       X::rend() const;
```

45 *Returns:* `const_reverse_iterator(begin())`

```
const_reverse_iterator X::crbegin() const;
```

46 *Returns:* `const_cast<const X&>(*this).rbegin()`

```
const_reverse_iterator X::crend() const;
```

47 *Returns:* `const_cast<const X&>(*this).rend()`

48 Unless otherwise specified (see **??** and 24.2.5.4) all container types defined in this clause meet the following additional requirements:

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

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

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

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

— no `swap()` function throws an exception unless that exception is thrown by the copy constructor or assignment operator of the container's Compare object (if any; see **??**).

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

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

24.1.1 Sequences [lib.sequence.reqmts]

1 A sequence is a kind of container that organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides three basic kinds of sequence containers: `vector`, `list`, and `deque`. It also provides container adaptors that make it easy to construct abstract data types, such as `stack`s or `queue`s, out of the basic sequence kinds (or out of other kinds of sequences that the user might define).

2 `vector`, `list`, and `deque` offer the programmer different complexity trade-offs and should be used accordingly. `vector` is the type of sequence that should be used by default. `list` should be used when there are frequent insertions and deletions from the middle of the sequence. `deque` is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.

3 ~~In Tables 82 and 83, X denotes a sequence class, a denotes a value of X, i and j denote iterators satisfying input iterator requirements, [i, j) denotes a valid range, n denotes a value of X::size_type, p denotes a valid iterator to a, q denotes a valid dereferenceable iterator to a, [q1, q2) denotes a valid range in a, and t denotes a value of X::value_type.~~

4 The complexities of the expressions are sequence dependent.

[[**Remove Table 82: Sequence requirements (in addition to container)**]]

5 Sequences are described by the `Sequence`, `FrontInsertionSequence`, and `BackInsertionSequence` concepts.

```
concept Sequence<typename X> : Container<X>
{
  where MutableForwardIterator<iterator> && ForwardIterator<const_iterator>;

  X::X(size_type n, value_type t);

  template<InputIterator Iter>
    where Convertible<Iter::value_type, value_type>
    X::X(Iter first, Iter last);

  iterator X::insert(iterator p, value_type t);
  void X::insert(iterator p, size_type n, value_type t);
  template<InputIterator Iter>
    where Convertible<Iter::value_type, value_type>
    void X::insert(iterator p, Iter first, Iter last);

  iterator X::erase(iterator q);
  iterator X::erase(iterator q1, iterator q2);

  void X::clear();

  template<InputIterator Iter>
    where Convertible<Iter::value_type, value_type>
    void X::assign(Iter first, Iter last);
  void X::assign(size_type n, value_type);
}
```

```
X::X(size_type n, value_type t);
```

6 *Effects:* constructs a sequence with `n` copies of `t`

7 *Postcondition:* `size() == n`

```
template<InputIterator Iter>
  where Convertible<Iter::value_type, value_type>
  X::X(Iter first, Iter last);
```

8 *Effects:* constructs a sequence equal to the range `[i, j)`

9 *Postconditions:* `size() == distance` between `i` and `j`

```
iterator X::insert(iterator p, value_type t);
```

10 *Effects:* inserts a copy of `t` before `p`

11 *Returns:* an iterator that points to the copy of `t` inserted into `a`.

```
void X::insert(iterator p, size_type n, value_type t);
```

12 *Effects:* inserts `n` copies of `t` before `p`

```
template<InputIterator Iter>
  where Convertible<Iter::value_type, value_type>
  void X::insert(iterator p, Iter first, Iter last);
```

13 *Precondition:* `i` and `j` are not iterators into `a`

14 *Effects:* inserts copies of elements in `[i, j)` before `p`

```
iterator X::erase(iterator q);
```

15 *Effects:* erases the element pointed to by `q`

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

```
iterator X::erase(iterator q1, iterator q2);
```

17 *Effects:* erases the elements in the range `[q1, q2)`.

18 *Returns:* ~~The iterator returned by a.erase(q1,q2)~~An iterator that points to the element pointed to by `q2` prior to any elements being erased. If no such element exists, `a.end()` is returned.

```
void X::clear();
```

19 *Effects:* `erase(begin(), end())`

20 *Postconditions:* `size() == 0`

```
template<InputIterator Iter>
  where Convertible<Iter::value_type, value_type>
  void X::assign(Iter first, Iter last);
```

21 *Precondition:* `i`, `j` are not iterators into `a`

22 *Effects:* Replaces elements in `a` with a copy of `[i, j)`.

```
void X::assign(size_type n, value_type);
```

23 *Precondition:* `t` is not a reference into `a`.

24 *Effects:* Replaces elements in `a` with `n` copies of `t`.

[[**Remove paragraphs 5–11, including the "do the right thing" clause.**]]

25 ~~Table 83 lists sequence operations that are provided for some types of sequential containers but not others. An implementation shall provide these operations for all container types shown in the "container" column, and shall implement them so as to take amortized constant time.~~The BackAccessSequence concept describes sequences for which the last element can be accessed in amortized constant time.

[[**Remove Table 83: Optional sequence operations**]]

```
concept BackAccessSequence<typename X> : Sequence<X>
{
  reference       X::back();
  const_reference X::back() const;
}
```

```
reference       X::back();
const_reference X::back() const;
```

26 *Returns:* `{ iterator tmp = end();`
`-tmp;`
`return *tmp; }`

27 The `BackInsertionSequence` concept describes sequences for which one can insert, remove, or access an element at the end of a container in amortized constant time.

```
concept BackInsertionSequence<typename X> : BackAccessSequence<X>
{
  void X::push_back(value_type x);
  void X::pop_back();
}
```

```
void X::push_back(value_type x);
```

28 *Effects:* `insert(end(),x)`

```
void X::pop_back();
```

29 *Effects:* `{ iterator tmp = end();`
`-tmp;`
`erase(tmp); }`

30 The `FrontAccessSequence` concept describes sequences for which one can access the element at the front of the container in amortized constant time.

```
concept FrontAccessSequence<typename X> : Sequence<X>
{
  reference       X::front();
  const_reference X::front() const;
}

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

31 *Returns:* `*begin()`

32 The `FrontInsertionSequence` concept describes sequences for which one can insert, remove, or access an element at the front of a container in amortized constant time.

```
concept FrontInsertionSequence<typename X> : FrontAccessSequence<X>
{
  void X::push_front(value_type x);
  void X::pop_front();
}

void X::push_front(value_type x);
```

33 *Effects:* `insert(begin(),x)`

```
void X::pop_front();
```

34 *Effects:* `erase(begin())`

35 The `RandomAccessSequence` concept describes sequences that provide access to any element in the container in amortized constant time.

```
concept RandomAccessSequence<typename X>
  : FrontAccessSequence<X>, BackAccessSequence<X>
{
  where MutableRandomAccessIterator<iterator> &&
        RandomAccessIterator<const_iterator>;

  reference       operator[](X& a, size_type n);
  const_reference operator[](const X& a, size_type n);

  reference       at(X& a, size_type n);
  const_reference at(const X& a, size_type n);
}

reference       operator[](X& a, size_type n);
const_reference operator[](const X& a, size_type n);
```

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

```
reference       at(X& a, size_type n);
const_reference at(const X& a, size_type n);
```

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

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

39 An implementation shall provide the following concept maps. When the implementation provides a vector<bool> specialization, vector<T> only meets the sequence concept when T is not bool.

```
template<CopyConstructible T, Allocator Alloc>
  where !SameType<T, bool> // iff vector<bool> specialization is provided
  concept_map RandomAccessSequence<vector<T, Alloc> > { }
template<CopyConstructible T, Allocator Alloc>
  where !SameType<T, bool> // iff vector<bool> specialization is provided
  concept_map BackInsertionSequence<vector<T, Alloc> > { }

template<CopyConstructible T, Allocator Alloc>
  concept_map BackInsertionSequence<list<T, Alloc> > { }
template<CopyConstructible T, Allocator Alloc>
  concept_map FrontInsertionSequence<list<T, Alloc> > { }

template<CopyConstructible T, Allocator Alloc>
  concept_map RandomAccessSequence<deque<T, Alloc> > { }
template<CopyConstructible T, Allocator Alloc>
  concept_map BackInsertionSequence<deque<T, Alloc> > { }
template<CopyConstructible T, Allocator Alloc>
  concept_map FrontInsertionSequence<deque<T, Alloc> > { }
```

24.2 Sequences [lib.sequences]

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

Header `<vector>` synopsis

```
namespace std {
  template <class T, Allocator Alloc = allocator<T> >
    where SameType<T, Alloc::value_type> class vector;
  template <class T, class Alloc>
    where EqualityComparable<T>
    bool operator==(const vector<T,Alloc>& x,
                    const vector<T,Alloc>& y);
  template <class T, class Alloc>
    where LessThanComparable<T>
    bool operator< (const vector<T,Alloc>& x,
                    const vector<T,Alloc>& y);
  template <class T, class Alloc>
    where EqualityComparable<T>
    bool operator!=(const vector<T,Alloc>& x,
                    const vector<T,Alloc>& y);
  template <class T, class Alloc>
    where LessThanComparable<T>
    bool operator> (const vector<T,Alloc>& x,
                    const vector<T,Alloc>& y);
  template <class T, class Alloc>
    where LessThanComparable<T>
    bool operator>=(const vector<T,Alloc>& x,
                    const vector<T,Alloc>& y);
  template <class T, class Alloc>
    where LessThanComparable<T>
    bool operator<=(const vector<T,Alloc>& x,
                    const vector<T,Alloc>& y);

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

24.2.4 Container adaptors [lib.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.

24.2.4.1 Class template queue [lib.queue]

1 Any ~~sequence supporting operations front(), back(), push_back() and pop_front()~~container that meets the requirements of the Back Insertion Sequence and Front Insertion Sequence concepts can be used to instantiate queue. In particular, list (??) and deque (??) can be used.

24.2.4.1.1 queue definition [lib.queue.defn]

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

  public:
    explicit queue(const Container& = Container());

    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 pop()                          { c.pop_front(); }
  };

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

24.2.4.1.2 `queue` operators [lib.queue.ops]

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

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

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

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

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

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

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

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

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

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

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

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

24.2.4.2 Class template `priority_queue` [lib.priority.queue]

1 Any sequence with random access iterator and supporting operations `front()`, `push_back()` and `pop_back()` can be used to instantiate `priority_queue`. In particular, `vector` (24.2.5) and `deque` (**??**) can be used. Instantiating `priority_queue` also involves supplying a function or function object for making priority comparisons; the library assumes that the function or function object defines a strict weak ordering (**??**).

```
  namespace std {
    template <class T, RandomAccessSequence Container = vector<T>,
             BinaryPredicate<Container::value_type, Container::value_type> Compare
               = less<Container::value_type> >
    where CopyConstructible<Compare>
    class priority_queue {
    public:
      typedef Container::value_type            value_type;
      typedef Container::reference             reference;
      typedef Container::const_reference       const_reference;
      typedef Container::size_type             size_type;
      typedef Container                        container_type;
    protected:
      Container c;
      Compare comp;

    public:
      explicit priority_queue(const Compare& x = Compare(),
                  const Container& = Container());
      template <InputIterator Iter>
        where Convertible<Iter::value_type, value_type>
        priority_queue(Iter first, Iter last,
               const Compare& x = Compare(),
               const Container& = Container());

      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 pop();
    };
                          // no equality is provided
  }
```

24.2.4.2.1 `priority_queue` constructors [lib.priqueue.cons]

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

1 *Requires:* x defines a strict weak ordering (**??**).

2 *Effects:* Initializes `comp` with `x` and `c` with `y`; calls `make_heap(c.begin(), c.end(), comp)`.

```
template <InputIterator Iter>
  where Convertible<Iter::value_type, value_type>
  priority_queue(Iter first, Iter last,
                 const Compare& x = Compare(),
                 const Container& y = Container());
```

3 *Requires:* x defines a strict weak ordering (**??**).

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

24.2.4.2.2 `priority_queue` members [lib.priqueue.members]

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

1 *Effects:*

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

```
void pop();
```

2 *Effects:*

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

24.2.4.3 Class template `stack` [lib.stack]

1 Any sequence supporting operations `back()`, `push_back()` and `pop_back()` can be used to instantiate `stack`. In particular, `vector` (24.2.5), `list` (**??**) and `deque` (**??**) can be used.

24.2.4.3.1 `stack` definition [lib.stack.defn]

```
  namespace std {
    template <class T, BackInsertionSequence Container = deque<T> >
    class stack {
```

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

  public:
    explicit stack(const Container& = Container());

    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 pop()                          { c.pop_back(); }
  };

  template <class T, class Container>
    where EqualityComparable<Container::value_type>
    bool operator==(const stack<T, Container>& x,
                    const stack<T, Container>& y);
  template <class T, class Container>
    where LessThanComparable<Container::value_type>
    bool operator< (const stack<T, Container>& x,
                    const stack<T, Container>& y);
  template <class T, class Container>
    where EqualityComparable<Container::value_type>
    bool operator!=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
  template <class T, class Container>
    where LessThanComparable<Container::value_type>
    bool operator> (const stack<T, Container>& x,
                    const stack<T, Container>& y);
  template <class T, class Container>
    where LessThanComparable<Container::value_type>
    bool operator>=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
  template <class T, class Container>
    where LessThanComparable<Container::value_type>
    bool operator<=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
}
```

24.2.4.3.2 `stack` operators [lib.stack.ops]

```
template <class T, class Container>
    where EqualityComparable<Container::value_type>
    bool operator==(const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

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

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

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

```
template <class T, class Container>
    where LessThanComparable<Container::value_type>
    bool operator< (const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

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

```
template <class T, class Container>
    where LessThanComparable<Container::value_type>
    bool operator<=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

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

```
template <class T, class Container>
    where LessThanComparable<Container::value_type>
    bool operator> (const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

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

```
template <class T, class Container>
    where LessThanComparable<Container::value_type>
    bool operator>=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
```

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

24.2.5 Class template `vector` [lib.vector]

1 A `vector` is a kind of sequence 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>` 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)~~ and of a sequence, including most of the optional sequence requirements (24.1.1). The exceptions are the `push_front` and `pop_front` member functions, which are not provided. ~~In addition to the requirements on the stored object described in 23.1, the stored object shall meet the requirements of Assignable.~~ Descriptions are provided here only for operations

on `vector` that are not described in one of these ~~tables~~concepts or for operations where there is additional semantic information.

```
namespace std {
  template <class T, ~~class~~Allocator Alloc = allocator<T> >
  where SameType<T, Alloc::value_type>
  class vector {
  public:
    // types:
    typedef typename Alloc::reference             reference;
    typedef typename Alloc::const_reference       const_reference;
    typedef implementation-defined                iterator;        // See 24.1
    typedef implementation-defined                const_iterator; // See 24.1
    typedef implementation-defined                size_type;       // See 24.1
    typedef implementation-defined                difference_type;// See 24.1
    typedef T                                     value_type;
    typedef Alloc                                 allocator_type;
    typedef typename Alloc::pointer               pointer;
    typedef typename Alloc::const_pointer         const_pointer;
    typedef std::reverse_iterator<iterator>       reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // 24.2.5.1 construct/copy/destroy:
    explicit vector(const Alloc& = Alloc());
    where CopyConstructible<T>
      explicit vector(size_type n, const T& value = T(), const Alloc& = Alloc());

    template <InputIterator Iter>
      where Convertible<Iter::value_type, T>
      vector(Iter first, Iter last, const Alloc& = Alloc());
    template <ForwardIterator Iter>
      where Convertible<Iter::value_type, T>
      vector(Iter first, Iter last, const Alloc& = Alloc());
    where CopyConstructible<T> vector(const vector<T,Alloc>& x);
   ~vector();
    where CopyConstructible<T> && Assignable<T>
      vector<T,Alloc>& operator=(const vector<T,Alloc>& x);
    template <InputIterator Iter>
      Convertible<Iter::value_type, T> && CopyConstructible<T> && Assignable<T>
      void assign(Iter first, Iter last);
    where CopyConstructible<T> && Assignable<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;

    // 24.2.5.2 capacity:
    size_type size() const;
    size_type max_size() const;
    where CopyConstructible<T> && Assignable<T> void resize(size_type sz, T c = T());
    size_type capacity() const;
    bool      empty() const;
    where CopyConstructible<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;

    // 24.2.5.3 data access
    pointer data();
    const_pointer data() const;

    // 24.2.5.4 modifiers:
    void push_back(const T& x);
    void pop_back();
    where CopyConstructible<T> && Assignable<T>
      iterator insert(iterator position, const T& x);
    where CopyConstructible<T> && Assignable<T>
      void insert(iterator position, size_type n, const T& x);
    template <InputIterator Iter>
      Convertible<Iter::value_type, T> && CopyConstructible<T> && Assignable<T>
      void insert(iterator position, Iter first, Iter last);
    template <ForwardIterator Iter>
      Convertible<Iter::value_type, T> && CopyConstructible<T> && Assignable<T>
      void insert(iterator position, Iter first, Iter last);
    where Assignable<T> iterator erase(iterator position);
    where Assignable<T> iterator erase(iterator first, iterator last);
    void     swap(vector<T,Alloc>&);
    void     clear();
  };

  template <class T, class Alloc>
    where EqualityComparable<T>
    bool operator==(const vector<T,Alloc>& x,
                    const vector<T,Alloc>& y);
  template <class T, class Alloc>
    where LessThanComparable<T>
```

```
      bool operator< (const vector<T,Alloc>& x,
                      const vector<T,Alloc>& y);
    template <class T, class Alloc>
      where EqualityComparable<T>
      bool operator!=(const vector<T,Alloc>& x,
                      const vector<T,Alloc>& y);
    template <class T, class Alloc>
      where LessThanComparable<T>
      bool operator> (const vector<T,Alloc>& x,
                      const vector<T,Alloc>& y);
    template <class T, class Alloc>
      where LessThanComparable<T>
      bool operator>=(const vector<T,Alloc>& x,
                      const vector<T,Alloc>& y);
    template <class T, class Alloc>
      where LessThanComparable<T>
      bool operator<=(const vector<T,Alloc>& x,
                      const vector<T,Alloc>& y);

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

24.2.5.1 `vector` constructors, copy, and assignment [lib.vector.cons]

```
vector(const Allocator& = Allocator());
where CopyConstructible<T>
  explicit vector(size_type n, const T& value = T(), const Allocator& = Allocator());
template <InputIterator Iter>
  where Convertible<Iter::value_type, T>
  vector(Iter first, Iter last, const Allocator& = Allocator());
template <ForwardIterator Iter>
  where Convertible<Iter::value_type, T>
  vector(Iter first, Iter last, const Allocator& = Allocator());
where CopyConstructible<T> vector(const vector<T,Allocator>& x);
```

1 *Complexity:* The constructor that accepts a forward iterator range 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~~The constructor that accepts an input iterator range makes order `N` calls to the copy constructor of `T` and order $\log(N)$ reallocations~~ if they are just input iterators~~.

```
template <InputIterator Iter>
  Convertible<Iter::value_type, T> && CopyConstructible<T> && Assignable<T>
  void assign(Iter first, Iter last);
```

2 *Effects:*

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

`where CopyConstructible<T> && Assignable<T> void assign(size_type n, const T& t);`

3 *Effects:*

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

24.2.5.2 `vector` capacity [lib.vector.capacity]

`size_type capacity() const;`

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

`where CopyConstructible<T> void reserve(size_type n);`

2 *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()`.

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

4 *Throws:* `length_error` if *n* `> max_size()`.[2)]

5 *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,Allocator>& x);`

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

7 *Complexity:* Constant time.

`where CopyConstructible<T> && Assignable<T> void resize(size_type sz, T c = T());`

8 *Effects:*

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

24.2.5.3 `vector` data [lib.vector.data]

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

2) `reserve()` uses `Allocator::allocate()` which may throw an appropriate exception.

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.

24.2.5.4 `vector` modifiers [lib.vector.modifiers]

```
where CopyConstructible<T> && Assignable<T>
  iterator insert(iterator position, const T& x);
where CopyConstructible<T> && Assignable<T>
  void insert(iterator position, size_type n, const T& x);
template <InputIterator Iter>
  Convertible<Iter::value_type, T> && CopyConstructible<T> && Assignable<T>
  void insert(iterator position, Iter first, Iter last);
template <ForwardIterator Iter>
  Convertible<Iter::value_type, T> && CopyConstructible<T> && Assignable<T>
  void insert(iterator position, Iter first, Iter last);
```

1 *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.

2 *Complexity:* If `first` and `last` are forward iterators~~bidirectional iterators, or random access iterators~~, the complexity is linear in the number of elements in the range `[first,last)` plus the distance to the end of the vector. If they are input iterators, the complexity is proportional to the number of elements in the range `[first,last)` times the distance to the end of the vector.

```
where Assignable<T> iterator erase(iterator position);
where Assignable<T> iterator erase(iterator first, iterator last);
```

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

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

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

24.2.5.5 `vector` specialized algorithms [lib.vector.special]

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

1 *Effects:*

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

24.2.6 Deprecated class vector<bool> [lib.vector.bool]

1 To optimize space allocation, a specialization of vector for bool elements may be provided:

```
namespace std {
  template <Allocator Alloc> class vector<bool, Alloc> {
  public:
    // types:
    typedef bool                                   const_reference;
    typedef implementation-defined                 iterator;        // See 24.1
    typedef implementation-defined                 const_iterator;  // See 24.1
    typedef implementation-defined                 size_type;       // See 24.1
    typedef implementation-defined                 difference_type;// See 24.1
    typedef bool                                   value_type;
    typedef Alloc                                  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 Alloc& = Alloc());
    explicit vector(size_type n, const bool& value = bool(),
                    const Alloc& = Alloc());
    template <InputIterator Iter>
      where Convertible<Iter::value_type, bool>
      vector(Iter first, Iter last, const Alloc& = Alloc());
    template <ForwardIterator Iter>
      where Convertible<Iter::value_type, bool>
      vector(Iter first, Iter last, const Alloc& = Alloc());
    vector(const vector<bool,Alloc>& x);
   ~vector();
    vector<bool,Alloc>& operator=(const vector<bool,Alloc>& x);
    template <InputIterator Iter>
      where Convertible<Iter::value_type, bool>
      void assign(Iter first, Iter last);
    template <ForwardIterator Iter>
      where Convertible<Iter::value_type, bool>
      void assign(Iter first, Iter 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;

    // 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(iterator position, const bool& x);
    void     insert (iterator position, size_type n, const bool& x);
    template <InputIterator Iter>
      Convertible<Iter::value_type, bool>
        void insert(iterator position, Iter first, Iter last);
    template <ForwardIterator Iter>
      Convertible<Iter::value_type, bool>
        void insert(iterator position, Iter first, Iter last);
    iterator erase(iterator position);
    iterator erase(iterator first, iterator last);
    void swap(vector<bool,Alloc>&);
    static void swap(reference x, reference y);
    void flip();                    // flips all bits
    void clear();
  };
```

```
    // specialized algorithms:
    template <Allocator Alloc>
      void swap(vector<bool,Alloc>& x, vector<bool,Alloc>& y);
  }
```

2 `reference` is a class that simulates the behavior of references of a single bit in `vector<bool>`.

3 The vector<bool> specialization meets all of the container and sequence requirements except that its iterators do not meet the ForwardIterator requirements.

Bibliography

[1] Douglas Gregor and Bjarne Stroustrup. Concepts (revision 1). Technical Report N2081=06-0151, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2006.