Document Number: WG21/N0966R1 X3J16/96-0148R1
Date: 11 July 1996
Project: Programming Language C++
Reply to: Matt Austern, Randy Smithey
austern@sgi.com, smithey@roguewave.com

Empty sections in clause 23

Motivation

Clause 23 contains requirements for generic containers, and also describes the classes `bitset`, `deque`, `list`, `queue`, `priority_queue`, `stack`, `vector`, `map`, `multimap`, `set`, and `multiset`. Many of the sections that describe these classes, however, are incomplete: the subclauses for `set`, `multiset`, and `multimap`, for example, document nothing other than `swap`. Several subclauses are missing entirely, and 24 subclauses are empty.

The amount of missing text is daunting. Fortunately, there is a shortcut: subclauses 23.1 [lib.container.requirements] document the behavior of generic containers, and those subclauses can be reused in the remainder of clause 23. `Vector`, for example, is a reversible sequence whose iterators are random access iterators, and that provides constant-time insertion and removal of elements at the end of the sequence but not at the beginning. This means that `vector` supports all of the operations in tables 75, 76, and 77, and some of the operations provided in table 78. The only things we have to do to document `vector` are to refer to the Sequence and Reversible Container requirements, specify which of the “optional sequence operations” in table 78 are supported, and describe any of `vector`’s operations that aren’t present in those tables or that have special semantics. In fact, then, the definition of `vector` in [lib.vector] is almost complete even though it appears to be almost empty! All that is missing is an explanation of why most of its members are undocumented in that section. There is still a fair amount of text to be added, but this shortcut makes the task much more manageable.

An additional problem is that class `bitset` is numbered as 23.2.1; it is thus a subsection of [lib.sequences], which describes STL sequences. This is incorrect and confusing: `bitset` is an encapsulation of bitmask operations, and has an interface that is completely unrelated to that of sequences. It should be moved to another location so that it is not a subsection of 23.2 (sequences) or 23.3 (associative containers).

Working paper changes

bitset

- Renumber [lib.template.bitset], currently numbered as 23.2.1, as 23.4.
- Move the header `<bitset>` synopsis from [lib.sequences] to [lib.template.bitset].

deque

- Add the following paragraph after paragraph 1 of [lib.deque]:

  A `deque` satisfies all of the requirements of a reversible container ([lib.container.requirements]) and of a sequence˝([lib.sequence.reqmts]), so it provides all operations described in Table 75 (Container requirements), Table˝76 (Reversible container requirements) and Table 77 (Sequence requirements).˝ Additionally, it provides all operations described in Table 78 (Optional sequence˝operations). 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.

- Add the following text at the beginning of [lib.deque.cons]

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

Effects: Constructs an empty `deque`, using the specified allocator.
Complexity: Constant.

```
explicit deque(size_t n, const T& value = T(),
               const Allocator& = Allocator());
```

Effects: Constructs a `deque` with `n` copies of `value`, using the specified allocator.
Complexity: Linear in `n`.

```
template<class InputIterator>
deque(InputIterator first, InputIterator last,
      const Allocator& = Allocator());
```

Effects: Constructs a `deque` equal to the range `[first, last)`, using the specified allocator.
Complexity: If the iterators `first` and `last` are forward iterators, bidirectional iterators, or random access iterators the constructor makes only N calls to the copy˝ constructor, and performs no reallocations, where N is `last - first`. It makes at most 2N calls to the copy constructor of `T` and logN reallocations if they are input iterators. [Footnote: The˝complexity is greater in the case of input iterators because each element must be added individually: it is˝impossible to determine the distance between `first` and `last` before doing the copying.]

```
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
```

Effects:

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

```
template <class Size, class T> void assign(Size n, const T& t =˝T());
```

Effects:

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

- Delete lib.deque.types, lib.deque.iterators, and lib.deque.access

list

- Add the following paragraph after paragraph 1 of [lib.list]:

  A `list` satisfies all of the requirements of a reversible container ([lib.container.requirements]) and of a sequence˝([lib.sequence.reqmts]), so it provides all operations described in Table 75 (Container requirements), Table˝76 (Reversible container requirements) and Table 77 (Sequence requirements.) A `list` also provides most operations described in Table 78 (Optional sequence˝operations). The exceptions are the `operator[]` and `at` member functions, which are not provided. [Footnote: These member member functions are only provided by containers whose˝iterators are random access iterators.] 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.

- Add the following text at the beginning of [lib.list.cons].

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

Effects: Constructs an empty list, using the specified allocator.
Complexity: Constant.

```
explicit list(size_type n, const T& value = T(),
              const Allocator& = Allocator());
```

Effects: Constructs a `list` with `n` copies of `value`, using the specified allocator.
Complexity: Linear in n.

```
template <class InputIterator>
list(InputIterator first, InputIterator last,
     const Allocator& = Allocator());
```

Effects: Constructs a `list` equal to the range `[first,last)`.
Complexity: Linear in `last - first.`

```
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
```

Effects:

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

```
template <class Size, class T> void assign(Size n, const T& t =˝T());
```

Effects:

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

- Delete lib.list.types, lib.list.iterators, and lib.list.access

vector

- Add the following paragraph after paragraph 1 of [lib.vector]:
  A `vector` satisfies all of the requirements of a reversible container ([lib.container.requirements]) and of a sequence˝([lib.sequence.reqmts]), so it provides all operations described in Table 75 (Container requirements), Table˝76 (Reversible container requirements) and Table 77 (Sequence requirements.) A `vector` also provides most operations described in Table 78 (Optional sequence˝operations). 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.

- Delete lib.vector.types, lib.vector.iterators, and lib.vector.access

map

- Add the following paragraph after paragraph 1 of [lib.map]:

  A **`map`** satisfies all of the requirements of a reversible container ([lib.container.requirements]) and of an associative container ([lib.associative.reqmts]), so it provides all operations described in Table 75 (Container requirements), and Table 76 (Reversible container requirements). A **`map`** also supports the requirements of Table 79 (Associative container requirements) for unique keys. This means that a **`map`** supports the **`a_uniq`** operations in Table 79, 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 these tables or for operations where there is additional semantic information.

- Add the following text at the beginning of [lib.map.cons]

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

Effects: Constructs an empty **`map`** using the specified comparsion object and allocator.
Complexity: Constant

```
template <class InputIterator>
map(InputIterator first, InputIterator last,
    const Compare& comp = Compare(), const Allocator& = Allocator());
```

Effects: Constructs an empty **`map`** using the specified comparison object and allocator, and inserts elements from the range **`[first,last)`**.
Complexity: Linear in N if the range **`[first,last)`** is already sorted using **`comp`** and otherwise NlogN, where N is **`last - first.`**

- Add the following paragraph in lib.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<const_iterator,const_iterator>
equal_range(const key_type& x) const;
pair<iterator,iterator> equal_range(const key_type& x);
```

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 versions is identical except that the const version returns a `const_iterator` and the non-const vesion an `iterator`. See Table 79 for a description of the behavior of these functions.

- Delete lib.map.types, lib.map.iterators, lib.map.capacity,˝lib.map.modifiers, lib.map.observers.

multimap

- Add the following paragraph after paragraph 1 of [lib.multimap]:

  A `multimap` satisfies all of the requirements of a reversible container ([lib.container.requirements]) and of an associative container˝([lib.associative.reqmts]), so it provides all operations described in Table 75 (Container˝requirements), and Table 76 (Reversible container requirements). A `multimap` also supports the requirements of Table 79 (Associative container requirements) for equal keys.˝ This means that a `multimap` supports the `a_eq` operations in Table 79, 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 these tables or for operations where˝there is additional semantic information.

- Add the following text as [lib.multimap.cons]

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

Effects: Constructs an empty `multimap` using the specified comparsion object and allocator.
Complexity: Constant.

```
template <class InputIterator>
multimap(InputIterator first, InputIterator last,
         const Compare& comp = Compare(),
         const Allocator& = Allocator());
```

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

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

- Add the following paragraph as lib.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;
```

```
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;
```

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 versions is identical except that the const version returns a `const_iterator` and the non-const vesion an `iterator`. See Table 79 for a description of the behavior of these functions.

set

- Add the following paragraph after paragraph 1 of [lib.set]:

  A `set` satisfies all of the requirements of a reversible container ([lib.container.requirements]) and of an associative container˝([lib.associative.reqmts]), so it provides all operations described in Table 75 (Container˝requirements), and Table 76 (Reversible container requirements). A `set` also supports the requirements of Table 79 (Associative container requirements) for unique keys. This˝means that a `set` supports the `a_uniq` operations in Table 79, but not the `a_eq` operations. For a `set<Key>` both the `key_type` and the `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.

- Add the following text in [lib.set.cons]

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

Effects: Constructs an empty set using the specified comparsion object and˝allocator.
Complexity: Constant.

```
template <class InputIterator>
set(InputIterator first, InputIterator last,
    const Compare& comp = Compare(), const Allocator& =˝Allocator());
```

Effects: Constructs an empty `set` using the specified comparison object and allocator, and inserts elements from the range `[first,last)`.
Complexity: Linear in N if the range `[first,last)` is already sorted using `comp` and otherwise NlogN, where N is `last - first.`

- Delete lib.set.types, lib.set.iterators, lib.set.capacity,˝lib.set.modifiers, lib.set.observers, and lib.set.ops.

multiset

- Add the following paragraph after paragraph 1 of [lib.multiset]:

  A `multiset` satisfies all of the requirements of a reversible container ([lib.container.requirements]) and of an associative container˝([lib.associative.reqmts]), so it provides all operations described in Table 75 (Container˝requirements), and Table 76 (Reversible container requirements). A `multiset` also supports the requirements of Table 79

(Associative requirements) for duplicate keys. This means that a˝`multiset` supports the `a_eq` operations in Table 79, but not the `a_uniq` operations. For a `multiset<Key>` both the `key_type` and the `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.

- Add the following text in[lib.multiset.cons]

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

Effects: Constructs an empty set using the specified comparsion object and˝allocator.
Complexity: Constant

```
template <class InputIterator>
multiset(InputIterator first, InputIterator last,
    const Compare& comp = Compare(), const Allocator& =˝Allocator());
```

Effects: Constructs an empty `multiset` using the specified comparison object and allocator, and inserts elements from the range `[first,last)`.
Complexity: Linear in N if the range `[first,last)` is already sorted using `comp` and otherwise NlogN, where N is `last - first.`

- Delete lib.multiset.types, lib.multiset.iterators,˝lib.multiset.capacity, lib.multiset.modifiers, lib.multiset.observers, and lib.multiset.ops.