Document No.:	J16/97-0064
	WG21/N1102
Project:	Programming Language C++
Date:	17 July, 1997
Reply To:	Steve Rumsby
	steve@maths.warwick.ac.uk

Other Library Changes from London

Steve Rumsby
steve@maths.warwick.ac.uk

Introduction

The plan in London was to produce all proposals in the form of working paper diffs. Due to time constraints not all of the proposals agreed on by the Library Group were converted to that format. Those are presented here in English.

Container Ambiguity

General Strategy

For each of the types basic_string<>, vector<>, vector<bool>, list<>, and deque<>, this affects the member template range versions of X<>::X() and X<>::insert(). (Those aren't the only functions that use ranges of iterators. They're sufficient, though, since all of the others are defined in terms of the constructor insert().)

We could just repeat something like:

> If InputIterator is an integral type, equivalent to insert(position, (size_type) begin, (value_type) end)

ten times, but that would be tedious and pointless. It's more sensible to state this equivalence once in, clause 23, since it's meant to apply uniformly, and then in clause 21.

There's also the issue of general requirements for sequences, defined in Table 68. We have to make it clear that for every conforming sequence, X(m, n) always means X((X::size_type) m, (X::value_type) n) if m and n are of an integral type. This should be a non-normative note: integral types are not iterators, so I believe it is already true that a container that tries to treat m and n as a range of iterators does not conform to this table's requirements. That fact isn't obvious, though, hence the note.

Clause 23 Changes

Add the following text after paragraph 8 of 23.1.1 [lib.sequence.reqmts].

```
For every sequence defined in this clause, the constructor
      template <class InputIterator>
    X(InputIterator f, InputIterator l, const Allocator& a = Allocator())

is, if InputIterator is an integral type, equivalent to

    X(static_cast<typename X::size_type>(f), static_cast<typename
X::value_type>(l), a)

and the member function

    template <class InputIterator>
    void insert(iterator p, InputIterator f, InputIterator l)

is, if InputIterator is an integral type, equivalent to

    insert(p, static_cast<typename X::size_type>(f), static_cast<typename
X::value_type>(l)).

[Note: This follows directly from the requirements in Table 68. Integral types
cannot be iterators, so, if n1 and n2 are values of an integral type N, the
expression X(n1, n2) cannot possibly be interpreted as construction from a range
of iterators. It must be taken to mean the first constructor in Table 68, not
the second one. If there is no conversion from N to X::value_type, then this is
not a valid expression at all.]
```

```
One way that sequence implementors can satisfy this requirement is to specialize
the member template for every integral type. Less cumbersome implementation
techniques also exist. --End Note]

[Example:

list<int> x;

...

vector<int> y(x.begin(), x.end());  // Construct a vector
                                    // from a range of iterators.

vector<int> z(100, 1);              // Construct a vector of 100
                                    // elements, all initialized
                                    // to 1. The arguments are
                                    // not interpreted as iterators.
--End Example]
```

Clause 21 Changes

Change the Effects: clause for the constructor

```
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end, const Allocator& a =
Allocator())
```

to:

```
Effects:
          If InputIterator is an integral type, equivalent to
                    basic_string(static_cast<size_type>(first),
                    static_cast<size_type>(last)).
          If InputIterator is an integral type, equivalent to
                    basic_string((size_type) begin, (value_type) end, a).
          Otherwise constructs a string from the values in the range
                    [begin, end), as indicated in Table 8 (see
                    [lib.sequence.reqmts]):
```

In 21.3.5.4 change the effects clause for

```
template<class InputIterator>
void insert(iterator p, InputIterator first, InputIterator last);
```

to

```
Effects:
          If InputIterator is an integral type, equivalent to
                    insert(p, static_cast<size_type>(first),
                    static_cast<size_type>(last)).
          Otherwise, inserts copies of the characters in the range
                    [first,last) before the character referred to by p.
          (see [lib.sequence.reqmts])
```

Exception Safety in the Library

Replace the last two sentences of 17.3.3.6, paragraph 2, with:

```
- for types used as template arguments when instantiating a template component,
if the operations on the type do not implement the semantics of the applicable
requirements subclause (20.1.5, 23.1, 24.1, 26.1). Operations on such types can
report a failure by throwing an exception unless otherwise specified.

- if any replacement function or handler function or destructor operation throws
an exception, unless specifically allowed in the applicable Required behavior
paragraph.
```

Replace paragraph 3 of section 17.3.4.8 with:

```
No destructor operation defined in the C++ Standard library will throw an
exception. Any other functions defined in the C++ Standard library that do not
```

```
have an exception-specification may throw implementation-defined exceptions
unless otherwise specified. 169) An implementation may strengthen this implicit
exception-specification by adding an explicit one. 170)
```

Add the sentence

```
Does not throw exceptions.
```

to the notes for a::deallocate() in Table 32 20.1.5.

Append to the last sentence of paragraph 1 in 20.4.4:

```
and is required to have the property that no exceptions are thrown from
increment, assignment, comparison, or dereference of valid iterators. In the
following algorithms, if an exception is thrown there are no effects.
```

Replace the Effects paragraph of 20.4.4.1 (uninitialized_copy) with:

```
Effects:
          for (; first != last; ++result, ++first)
                    new (static_cast<void*>(&*result))
                    typename
                    iterator_traits<ForwardIterator>::value_type(*first);
```

Replace the Effects paragraph of 20.4.4.2 (uninitialized_fill) with:

```
Effects:
          for (; first != last; ++first)
                    new (static_cast<void*>(&*first))
                    typename iterator_traits<ForwardIterator>::value_type(x);
```

Replace the Effects paragraph of 20.4.4.3 (uninitialized_fill_n) with:

```
Effects:
          for (; n--; ++first)
                    new (static_cast<void*>(&*first))
                    typename iterator_traits<ForwardIterator>::value_type(x);
```

Append the following sentence to 21.1.2, paragraph 1:

```
If any operation on Traits throws an exception the effects are undefined.
```

Append the following paragraph to section 23.1 (Container requirements):

```
All container types defined in this clause meet the additional requirements that
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 23.1.2).
```

Insert the following signatures before paragraph 1 of section 23.2.2.3 (list::insert):

```
void push_front(const T& x);
void push_back(const T& x);
```

Append the following sentence to paragraph 1 of section 23.2.2.3 (list::insert):

```
If an exception is thrown there are no effects.
```

Insert the following signatures before paragraph 3 of section 23.2.2.3: (list::erase):

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

Insert the following paragraph after paragraphs 4 of section 23.2.2.3 (list::erase):

```
Throws: Nothing.
```

Insert the following paragraph after paragraphs 4, 6 & 9 of section 23.2.2.4 (list::splice):

```
Throws: Nothing.
```

Insert the following paragraph after paragraph 12 of section 23.2.2.4 (list::remove and list::remove_if):

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

Insert the following paragraph after paragraph 15 of section 23.2.2.4 (list::unique):

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

Insert the following paragraph after paragraph 21 of section 23.2.2.4 (list::reverse):

```
Throws: Nothing.
```

Append the following sentence to paragraph 19 of section 23.2.2.4 (list::merge):

```
If an exception is thrown other than by a comparison there are no effects.
```

Append the following sentence to paragraph 25 of section 23.2.2.4 (list::sort):

```
If an exception is thrown the order of the elements in the list is
indeterminate.
```

Insert the following paragraph after 26.1, paragraph 1:

```
If any operation on T throws an exception the effects are undefined.
```

Constraints on Signal Handlers

Add the following text somewhere in section 18.7 [lib.support.runtime]:

```
The common subset of the C and C++ languages consists of all declarations,
definitions, and expressions which may appear in a well formed C++ program and
also in a conforming C program.

A POF ("plain old function") is a function which uses only features from this
common subset, and which does not directly or indirectly use any function that
is not a POF.

All signal handlers shall have C linkage.

A POF which could be used as a signal handler in a conforming C program does not
produce undefined behavior when used as a signal handler in a C++ program.\f*
.Fs
In particular, a signal handler using exception handling is very likely to have
problems
.Fe

The behavior of any other function used as a signal handler in a C++ program is
implementation defined.
```

Object Identity for Iterators

Add the following text immediately before paragraph 2 of 24.1.3 [lib.forward.iterators], that is, immediately after the table of forward iterator requirements.

```
-- If a and b are equal, then either a and b are both dereferenceable or else
neither is dereferenceable.

-- If a and b are both dereferenceable, then a == b if and only if *a and *b are
the same object.
```

Assignability of Insert Iterators

These iterators must now be assignable, and hence the existing reference member "container" must be changed to a pointer. The changed lines are indicated in *bold italics*, and consist of 12 new characters.

```
24.4.2.1 Template class back_insert_iterator
           namespace std {
                      template <class Container>
```

```
class back_insert_iterator : public
iterator<output_iterator_tag,void,void,void,void> {

protected:
        Container* container;
public:
        typedef Container container_type;
        explicit back_insert_iterator(Container& x);
        back_insert_iterator<Container>&
        operator=(const typename Container::value_type&
        value);
        back_insert_iterator<Container>& operator*();
        back_insert_iterator<Container>& operator++();
        back_insert_iterator<Container>
        operator++(int);
};

template <class Container>
back_insert_iterator<Container> back_inserter(Container&
x);
}
```

24.4.2.2 back_insert_iterator operations

24.4.2.2.1 back_insert_iterator constructor

```
explicit back_insert_iterator(Container& x);
```

Effects:

Initializes container with &x.

24.4.2.2.2 back_insert_iterator::operator=

```
back_insert_iterator<Container>&
operator=(const typename Container::value_type& value);
```

Effects:

container->push_back(value);

Returns:

*this.

24.4.2.2.3 back_insert_iterator::operator*

```
back_insert_iterator<Container>& operator*();
```

Returns:

*this.

24.4.2.2.4 back_insert_iterator::operator++

```
back_insert_iterator<Container>& operator++();

back_insert_iterator<Container> operator++(int);
```

Returns:

*this.

24.4.2.2.5 back_inserter

```
template <class Container>
back_insert_iterator<Container> back_inserter(Container& x);
```

Returns:

back_insert_iterator<Container>(x).

24.4.2.3 Template class front_insert_iterator

```
namespace std {
    template <class Container>
    class front_insert_iterator : public
    iterator<output_iterator_tag,void,void,void,void> {

    protected:
            Container* container;
    public:
            typedef Container container_type; explicit
            front_insert_iterator(Container& x);
            front_insert_iterator<Container>&operator=(const
            typename Container::value_type& value);
            front_insert_iterator<Container>& operator*();
            front_insert_iterator<Container>& operator++();

            front_insert_iterator<Container>
            operator++(int);
```

```
    };

    template <class Container>
    front_insert_iterator<Container> front_inserter(Container&
    x);
}
```

24.4.2.4 front_insert_iterator operations

24.4.2.4.1 front_insert_iterator constructor

```
explicit front_insert_iterator(Container& x);
```

Effects:

Initializes container with &x.

24.4.2.4.2 front_insert_iterator::operator=

```
front_insert_iterator<Container>&
operator=(const typename Container::value_type& value);
```

Effects:

container->push_front(value);

Returns:

*this.

24.4.2.4.3 front_insert_iterator::operator*

```
front_insert_iterator<Container>& operator*();
```

Returns:

*this.

24.4.2.4.4 front_insert_iterator::operator++

```
front_insert_iterator<Container>& operator++();
front_insert_iterator<Container> operator++(int);
```

Returns:

*this.

24.4.2.4.5 front_inserter

```
template <class Container>
front_insert_iterator<Container> front_inserter(Container& x);
```

Returns:

front_insert_iterator<Container>(x).

24.4.2.5 Template class insert_iterator

```
namespace std {
    template <class Container>

    class insert_iterator : public
    iterator<output_iterator_tag,void,void,void,void> {

    protected:
        Container* container;
        typename Container::iterator iter;
    public:
        typedef Container container_type;
        insert_iterator(Container& x, typename
        Container::iterator i);
        insert_iterator<Container>&
        operator=(const typename Container::value_type&
        value);
        insert_iterator<Container>& operator*();
        insert_iterator<Container>& operator++();
        insert_iterator<Container>& operator++(int);
    };

    template <class Container, class Iterator>
    insert_iterator<Container> inserter(Container& x, Iterator
    i);
}
```

24.4.2.6 insert_iterator operations

24.4.2.6.1 insert_iterator constructor

```
insert_iterator(Container& x, typename Container::iterator i);
```

Effects:

Initializes container with &x and iter with i.

```
24.4.2.6.2 insert_iterator::operator=
           insert_iterator<Container>&
           operator=(const typename Container::value_type& value);

           Effects:
                      iter = container->insert(iter, value);
                      ++iter;
           Returns:
                      *this.
24.4.2.6.3 insert_iterator::operator*
           insert_iterator<Container>& operator*();

           Returns:
                      *this.
24.4.2.6.4 insert_iterator::operator++
           insert_iterator<Container>& operator++();
           insert_iterator<Container>& operator++(int);

           Returns:
                      *this.
24.4.2.6.5 inserter
           template <class Container, class Inserter>
           insert_iterator<Container> inserter(Container& x, Inserter i);

           Returns:
                      insert_iterator<Container>(x,typename
                      Container::iterator(i)).
```